НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет Інформатики та Обчислювальної Техніки _____
(повна назва інституту/факультету)

Автоматики і Управління в Технічних Системах_____
(повна назва кафедри)

«На правах рукопису»
УДК 004.4.28_____

«До захисту допущено»

Завідувач кафедри

_____

_____
(підпис) (ініціали, прізвище)

“___”_____20__
р.

Магістерська дисертація

зі спеціальності (спеціалізації) 121 Інженерія програмного забезпечення
(код і назва спеціальності)

_____

на тему: Бібліотека швидкої розробки програмного забезпечення_____

_____

Виконав (-ла): студент (-ка) _6__ курсу, групи __ІТ-73мп__
(шифр групи)

Троцький Максим Олегович _____ _____
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник доцент каф. АУТС, к.т.н., доц.. Дорогий Я.Ю. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант _____ _____ _____
(назва розділу) (науковий ступінь, вчене звання, , прізвище, ініціали) (підпис)

Рецензент доцент каф. ІБ, к.т.н., доц. Демчинський В.В. ____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2018 року

РЕФЕРАТ

Магістерська дисертація присвячена бібліотеці швидкої розробки програмного забезпечення, а саме: бібліотеці контекстуальної валідації.

Магістерська дисертація містить 95 аркушів пояснювальної записки, 6 рисунків, 49 таблиць, 8 креслеників та 22 бібліографічних посилань на використані літературні джерела.

Актуальність даної роботи полягає в тому, що значна частина коду та, відповідно, потраченого на нього робочого часу, є реалізацією валідаційної логіки, зокрема контекстуальної. Проте, не зважаючи на поширеність проблеми, популярні фреймворки надають бідний інтерфейс для реалізації контекстуальної валідації.

Мета дисертації – обґрунтування необхідності нового підходу написання логіки контекстуальної валідації, опис авторського рішення – спеціалізованої бібліотеки – у розрізі існуючих рішень та постулатів дизайну програмного забезпечення.

Об’єктом дослідження виступає програмно-апаратна система моніторингу стану будівлі, у якій використовується дана бібліотека. Предметом дослідження є зміна метрик якості дизайну коду (Code Smells, Design Smells, Coupling / Coherency) системи внаслідок використання бібліотеки контекстуальної валідації.

Елементом науково-практичної новизни у даній роботі виступає бібліотека контекстуальної валідації. Її використання сприяє зменшенню об’єму коду, цикломатичної складності, позитивному впливу на дизайн застосунку та пришвидшенню розробки ПЗ.

За результатами проведеної роботи було написано статтю «Бібліотека контекстуальної валідації у середовищі .NET», яка була опублікована у збірнику матеріалів II Міжнародної науково-практичної конференції «Інтеграція світових наукових процесів як основа суспільного прогресу».

Ключові слова: .NET, валідація, domain driven design, code smells, design smells, coupling, coherency.

SUMMARY

The master's dissertation is devoted to the library of rapid software development, namely: the library of contextual validation.

The master's dissertation contains 95 sheets of explanatory note, 6 pictures, 49 tables, 8 drawings and 22 bibliographic references on used literary sources.

The relevance of this work lies in the fact that a significant part of the code and, accordingly, spent on it working time, is the implementation of validation logic, in particular, contextual. However, despite the prevalence of the problem, popular frameworks provide a poor interface for implementing contextual validation.

The purpose of the dissertation is to substantiate the need for a new approach to writing the contextual validation logic, provide description of the author's decision (the specialized library) in the context of existing decisions and software design principles.

The object of research is the software-hardware system for monitoring the state of the building in which the library is used. The subject of the study is the change of code design quality metrics (Code Smells, Design Smells, Coupling / Coherency) of the system as a result of using the Contextual Validation Library.

An element of scientific and practical novelty in this work is the Library of Contextual Validation. Its use reduces the volume of the code and cyclical complexity, has a positive impact on the design of the application and speeds up the development of software.

According to the results of the work, an article "Library of Contextual Validation in .NET." was written and published in the collection of materials of the 2nd International Scientific and Practical Conference "Integration of World Scientific Processes as the Basis of Social Progress".

Keywords: .NET, validation, domain driven design, code smells, design smells, coupling, coherency.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

SPA – Single Page Application

SSO – Single Sign On

WCF – Windows Communication Foundation

ASP – Active Server Pages

SOLID – Single Responsibility Principle, Open-Close Principle, Liskov Substitute Principle, Interface Segregation Principle, Dependency Inversion Principle

СКБД – Система керування базами даних

Id – Ідентифікатор

PK – обмеження цілісності Primary Key

UQ – обмеження цілісності Unique

NN – обмеження цілісності Not Null

FK (xxx) – обмеження цілісності Foreign Key (назва таблиці)

ORM – Object Relational Mapping

OWIN – Open Web Interface For .NET

DDD – Domain Driven Design

JWT – JSON Web Token

ВСТУП

На сьогоднішній день практика написання корпоративних додатків (зокрема і у середовищі .NET) показує, що від 10 до 40 відсотків коду та, відповідно, потраченого на нього робочого часу (що у свою чергу впливає на швидкість розробки ПЗ, а отже і на його собівартість та конкурентоспроможність компанії на ринку outsource development) витрачається на валідацію (перевірку) бізнес-правил.

Для спрощення та пришвидшення даного процесу використовуються валідаційні інструменти. Наявні наразі фреймворки чи бібліотеки для валідації у середовищі .NET у повній мірі реалізують лише позаконтекстуальну валідацію (на основі атрибутів чи Fluent API). Проте пристойний об’єм логіки валідації є саме контекстуальним (результат валідації залежить від інших частин програми, а не самої сутності), що здебільшого передбачає взаємодію із репозеторієм або іншим джерелом персистентності (у тому числі і в CQRS). Отже, весь необхідний для цього код пишеться розробниками стандартним імперативним підходом і, як правило, безпосередньо у модулях функціоналу ПЗ.

Задача полягає у тому, щоб виокремити логіку контекстуальної валідації від решти бізнес логіки, що зменшить зв’язаність програми, підвищить її спроможність тестування, а декларативний стиль подачі підвищить підтримуваність та ясність коду.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис проблематики

Валідація [1] є широкою темою, оскільки вона поширена у всіх областях застосування (application). Валідацію важко реалізувати на практиці, оскільки вона повинна бути реалізована у всіх областях застосування, як правило, з використанням різних методів для кожної області. У загальному сенсі, валідація – це механізм верифікаційних операцій, що встановлюють дійсність станів. Не слід випускати з уваги двозначність в цьому твердженні, оскільки вона ілюструє декілька важливих характеристик перевірки. Одна з характеристик - контекст, у якому викликається перевірка. Контекст має вирішальне значення, оскільки валідація в одному контексті може бути не прийнятною в іншому контексті. Іншим наслідком є неоднозначність того, що вважається дійсним. Дійсність може бути задана як відповідність тривіальному логічному твердженню, такому як «Строка, що представляє ім'я клієнта, не повинна бути null», або ж як складна послідовність тверджень CycL.

У даній роботі валідація трактується як маніфест та інваріанти у корпоративних додатках на базі DDD, звичайної N-layer та інших архітектур.

1.1.1 Основні засади організації валідаційної логіки

У доменному дизайні поширені дві школи думки [1], що стосуються валідації, які обертаються навколо поняття завжди валідного об'єкта. Джеффрі Палермо (відомий архітектор застосунків, автор дизайну Onion Architecture) стверджує, що завжди дійсна сутність – це омана. За його публікаціями, [2] логіка перевірки повинна бути відокремлена від об'єкту, що відповідно відклало би визначення правил перевірки. Інша школа думки [3], підтримана Грегом Яном (архітектор, автор підходу Design by Contract) та іншими, стверджує, що сутності повинні бути валідними завжди.

1.1.1.1 Валідація всередині сутності

Школа завжди валідної сутності ґрунтується на постулатах Domain Driven Design. З точки зору DDD правила перевірки можна розглядати як інваріанти – предикати, що встановлюються у самих типах моделі та є незмінними у продовж всього часу виконання програми. Одним із центральних обов'язків агрегату (а також, часто, і простої сутності) – кореневого типу моделі – є примусове застосування інваріантів при зміні стану об’єкту.

Якщо розглядати поділ команд, запитів і закриття операцій не тільки на об'єктах сервісів, а й на об'єктах-сутностях, у клієнтському коді можна ставитися до об'єктів із більшою довірою і не перетягувати їх специфічну логіку в ті області, до яких вона дійсно не належить. Вимога для виклику перевірки або запиту властивості на кшталт «IsValid» вимагає, щоб код виклику був неатомарний, і це може привести до неузгодженості і більшої імовірності людської помилки. Простіше кажучи, якщо контролювати операційну сторону валідації у самих сутностях (всередині сетерів властивостей, конструкторів та методів, що змінюють стан об’єкту), яким чином вони можуть потрапити в неприпустимий стан? Життя стає набагато складніше, якщо мати властивості «IsValid» на сутностях або сторонніх типах-валідаторах. Більш того, винесення логіки забезпечення правильності стану об’єкту за його межі призводить до поступового збіднення моделі, перетворення її на анемічну [4].

Перевагою такого підходу є те, що розробник типу повністю встановлює межі допустимого використання і клієнтський код взагалі не в змозі змінити стан об’єкту на такий, що суперечить бізнес-правилам. Проте таким чином зовсім ігнорується контекст виконання операції, або, інакше кажучи, даний підхід задає один і тільки один контекст виконання у будь-який момент часу.

1.1.1.2 Зовнішня валідація

Натомість у реальних системах існує складна логіка, яка вирішує, коли суб'єкт є дійсним для конкретних операцій, а коли ні. Існує рідко тільки один контекст для того, щоб бути дійсним чи недійсним. Наведу приклад. У системі використовується клас UserInfo, що відповідає інформації про користувача. Нехай із часом до типу додалася властивість Age, що описує вік користувача. За

вимогами бізнес-логіки вік особи завжди повинен бути присутнім в об’єкті. Проте при завантаженні старих даних вік можуть бути відсутнім. Чи вважаються такі дані невалідними? Чи повинна програма викинути виключення при їх завантаженні? Звісно, через описані зміни класу UserInfo система не повинна приходити у недійсний стан. Таким чином постають два контексти – створення нових та завантаження старих користувачів.

Також можливі випадки, коли контекст валідації в принципі не прив’язаний до стану одного конкретного об’єкту. Наприклад, неможна встановити значення поля Discount на об’єкті UserInfo, якщо сума покупок із відповідним user_info_id менша за певне значення. При цьому за отримання даної інформації відповідає тип OrderRepository, а за зберігання – Order – абсолютно інші типи. У цьому випадку знову постає проблема контексту виконання валідації.

1.1.1.3 Висновок

Отже, не всі правила перевірки можна представити у вигляді інваріантів, які імплементуються у самій сутності, важливо і необхідно враховувати контекст валідації. Таким чином постає теоретична обумовленість створення необхідних засобів.

1.1.2 Практики дизайну валідаційної логіки

Існує кілька способів реалізації валідації, таких як перевірка даних та викид винятків при невідповідності бізнес-правилам. Існують також більш досконалі підходи, такі як використання шаблону специфікації для перевірок, і шаблон сповіщення для повернення набору помилок, замість того, щоб повертати виняток для кожної перевірки, по ходу її порушення. У своєму офіційному керівництві «.NET Microservices Architecture» [5] Microsoft описують «best-practices» організації перевірки дійсності стану сутності. Ці офіційні рекомендації для середовища .NET описані нижче.

1.1.2.1 Викидання виключень

Даний підхід полягає у послідовній перевірці параметрів, що змінюють стан об’єкту у сетерах полів, методах та конструкторі. Якщо значення деякого

атрибуту сутності недійсне, а частина атрибутів пройшли валідацію та змінили стан сутності, це може зробити всю сутність недійсною. Тому при знаходженні першого невідповідного бізес-правилам параметра викидається виняток. Це запобігає переходу сутності у невалідний стан, проте надає інформацію лише про перше знайдене порушення. Даний підхід акомпонує поняттю «Always valid entity».

1.1.2.2 Використання атрибутів анотацій даних [6]

Інший підхід полягає у використанні атрибутів перевірки на основі анотацій даних. Атрибути перевірки забезпечують спосіб налаштування перевірки моделі, яка концептуально аналогічна до перевірки на полях в таблицях баз даних. Це включає обмеження, такі як призначення типів даних або обов'язкових полів. Інші типи перевірки включають застосування патернів даних для дотримання бізнес-правил, таких як номер кредитної картки, номер телефону або електронна адреса. Атрибути перевірки спрощують виконання вимог.

Самі по собі атрибути не забезпечують дійсності моделі. Анотації існують лише у вигляді метаданих типу і для перевірки правильності стану об’єкту необхідно через механізм рефлексії завантажити у пам'ять атрибути та вичитати обмеження, які вони задають. Самотужки такий код писати не потрібно, оскільки .NET надає клас Validator зі статичним методом TryValidateObject, який виконує необхідні операції. Більше того, анотації даних підтримуються ASP.NET, Entity Framework та іншими фреймворками і значно полегшують валідацію даних через автоматизацію значної частини роботи. Саме тому даний підхід набув широкого поширення. Проте в останніх публікаціях Microsoft не рекомендує реалізовувати його у шарі доменної логіки, а також відмовилась від його підтримки в Entity Framework Core 2.0 і вище.

1.1.2.3 Імплементація IValidatableObject [6]

Реалізація методу Validate, в об’єкті, що імплементує інтерфейс IValidatableObject, ідеологічно схожа на написання властивості IsValid, проте має суттєві відмінності. По-перше, Validate повертає перелік ValidationResult, що дає можливість не тільки встановити валідність сутность, проте й, у разі порушення

інваріантів, встановити причину. По-друге, метод Validate приймає контекст валідації, який по-суті являє собою загальний IServiceProvider, тобто, хоча і існує можливість контекстуальної валідації, код на всі контекст буде один, що призведе до підвищеної цикломатичної складності програми, до того ж, уся логіка перевірок пишеться вручну імперативним підходом, а не декларативно як в атрибутах. Робота з IValidatableObject підтримується середовищем .NET так само як і анотації даних, проте менш розповсюджена.

1.1.2.4 Notification патерн

У найпростішому вигляді Notification [7] може бути лише сукупністю рядків, які є повідомленнями про помилки, що генерує домен, поки він виконує свою роботу. При кожній перевірці, яку робить шар домену, невідповідність інваріанту призводить до появи помилки в сповіщенні. Проте має сенс надати сповіщенню більш явний інтерфейс. Об'єкт повідомлення, як правило, має методи додавання помилок, які використовують коди помилок, а не рядки, а також методи, щоб визначити, чи в повідомленні є якісь помилки. Таким чином надається вичерпний перелік порушених бізнес-правил. Даний підхід також не відповідає концепції «Always valid entity», та не має готової реалізації у середовищі .NET, проте його реалізація є хорошим архітектурним стилем

1.1.2.5 Specification патерн

Специфікація [8] – це особливий шаблон проектування програмного забезпечення, завдяки якому бізнес-правила можуть рекомбінуватися шляхом зчеплення з іншими бізнес-правилами за допомогою булевих операторів. Таким чином, логіка перевірки інкапсулюються в одному типі (не сутності, а окремому типі), що акомпонує правилу DRY. Через свої властивості підхід набув широкого розповсюдження, проте підтримка середовищем .NET відсутня. Не зважаючи на відповідність DDD, використання даного патерну не забезпечує реалізацію завжди валідної сутності.

1.1.2.6 Висновок

Вище перелічені підходи мають свої переваги та недоліки, проте всі вони активно застосовуються в організації логіки перевірки бізнес-правил.

Рекомендовані Microsoft підходи або неповноцінно, або взагалі не реалізують контекстуальну валідацію. Таким чином, постає вимога створення засобу контекстуальних верифікацій станів, який однаково успішно працював би з кожним із описаних прийомів.

1.2 Огляд існуючих рішень

Валідація може бути реалізована за допомогою тривіальних потоків управління if-then, проте такий код погано читається та підтримується, має високу цикломатичну складність. Саме тому широкої популярності набули валідаційні фреймворки, які насамперед надають можливість імперативного задання бізнес-правил та роботи з ними.

Таблиця 1.1 – Рейтинг бібліотек валідації за популярністю [9]

Назва	Кількість завантажень	Дата останнього оновлення
System.ComponentModel.DataAnnotations	35 млн	29.05.2018
FluentValidation	8,35 млн	04.09.2018
NHibernate Validator	86,2 тис.	02.04.2018
FubuValidation	71,9 тис.	08.09. 2014

Як видно з таблиці 1.1, популярністю користуються, по суті, лише два валідаційних фреймворка – FluentValidation (outsource проект на чолі із Джеремі Скіннером) та System.ComponentModel.DataAnnotations – розробка Microsoft, частина екосистеми .NET. З огляду на рейтинг, у ході розробки бібліотеки за основу бралися перші два фреймворки.

1.2.1 System.ComponentModel.DataAnnotations

1.2.1.1 Загальний опис

Простір імен System.ComponentModel.DataAnnotations містить класи атрибутів, які використовуються для визначення метаданих для керування даними

в ASP.NET MVC та ASP.NET, проте також можна з легкістю самому реалізувати код, що працює з цими атрибутами.

ValidationAttribute [10] – служить як базовий клас для всіх атрибутів перевірки та задає ключові функціональні точки:

– вказує, чи потрібен атрибуту контекст перевірки;

– задає повідомлення про помилку, у разі порушення інваріанту;

– перевіряє, чи вказане значення є дійсним щодо поточного атрибута валідації;

– видає повідомлення про помилку;

– валідує із урахуванням заданого контексту (який реалізує тип ValidationContext).

Його розширюють декілька атрибутів, що задають готові правила перевірки або надають більш зручний інтерфейс для їх задання.

Таблиця 1.2 – Базові атрибути валідації System.ComponentModel. DataAnnotations [6]

Назва	Функція
CreditCardAttribute	Значення перевіряється за допомогою регулярного виразу. Клас не підтверджує, що номер кредитної картки дійсний для покупок, лише що він відповідає формату.
EmailAddressAttribute	Валідація адреси електронної пошти на відповідність формату за допомогою регулярних виразів
FileExtensionsAttribute	Валідує розширення назви файлу на відповідність відомих, або явно заданих форматів
MaxLengthAttribute, MinLengthAttribute	Визначають відповідно максимальну або мінімальну довжину масивів або рядків даних, дозволених у властивості

Продовження таблиці 1.2.

Назва	Функція
PhoneAttribute	Валідує, що значення поля даних є добре номером телефону, що відповідає регулярному виразу
RangeAttribute	Визначає обмеження чисельного діапазону для значення поля даних
RegularExpressionAttribute	Валідує поле на відповідність явно заданому регулярному виразу
RequiredAttribute	Вказує, що значення поля даних не може бути null

Окрім готових валідаторів, даний простір імен також надає можливість створювати власні атрибути. Так, CustomValidationAttribute використовується для виконання користувацької перевірки, коли метод IsValid викликається для виконання перевірки. Метод IsValid переспрямовує виклик на метод, ідентифікований властивістю Method, який у свою чергу виконує фактичну перевірку.

Атрибут CustomValidationAttribute можна застосувати до типів, властивостей, полів, методів та параметрів методу. Коли він застосовується до властивості, атрибут викликається кожного разу, коли для цього ресурсу присвоюється значення. Коли він застосовується до методу, атрибут викликається кожного разу, коли програма викликає цей метод. Коли він застосовується до параметра методу, атрибут викликається до виклику методу.

Окрім валідації за допомогою метаданих (атрибутів), простір імен надає можливість створення сутності, яка валідує сама себе. Для цього типу сутності достатньо лише імплементувати інтерфейс IValidatable із єдиним методом Validate, що аналогічно до атрибутів приймає ValidationContext та повертає перелік ValidationResult.

Статичний клас Validator надає методи (Try)ValidateObject / Property / Value, у яких власне і виконується валідація як типів, що помічені атрибутами, так і таких, що реалізують вищеописаний інтерфейс. Саме ці методи викликаються за

кулісами в ASP.NET Framework під час визначення властивості ModelState.IsValid і не тільки.

1.2.1.2 Переваги

System.ComponentModel.DataAnnotations є найбільш поширеним і добре відомим широкому загалу розробників рішенням. Це, на пару із підтримкою самою платформою є основними козирями даного засобу організації валідаційної логіки. Загалом, можна виділити такі переваги:

– усі правила перевірки можна налаштувати в одному місці в коді (у межах класу метаданих моделі) і не потрібно повторювати де-небудь ще;
– відмінна підтримка перевірки на стороні клієнта;
– декларативний синтаксис;
– широкий спектр готових рішень;
– можливість розширення набору валідаційних атрибутів;
– інтеграція в екосистему .NET;
– широка спільнота: існують додаткові атрибути перевірки, створені спільнотою (наприклад, розширення анотацій даних);
– гарна задокументованість.

1.2.1.3 Недоліки

Рефлексивна природа атрибутів має ряд вад, насамперед, відсутність простого способу написання тестів коду, що забезпечує виконання інваріантів. Це призводить до того, що ця логіка не покривається тестами зовсім, що у свою чергу тягне за собою регресії під час рефакторингу. Ще одним суттєвим недоліком є неповноцінна підтримка контексту валідації, який у реалізації ValidationContext є не набагато більше ніж огортка над Object типом. Це негативно впливає на безпеку типів. Також, написання єдиного коду для декількох контекстів приведе до порушення принципів SOLID [11], та виникнення Code Smells (в першу чергу, Bloaters та Object-Oriented Abusers).

Загалом, можна виділити такі недоліки:

– неможливість реалізації «Always valid entity»;

– неможливість реалізації Specification патерну;

– складність реалізації Notification патерну через рефлексивну природу атрибутів;

– складність задання валідації властивості відносно значення іншої властивості;

– складність покриття валідаційної логіки юніт-тестами;

– неможливість виконання валідації по частковій виборці інваріантів;

– відсутність поліморфізму правил перевірки;

– неявна зв’язність логіки валідації із сутності від якої неможливо позбутися, і яка при цьому не забезпечує «Always valid entity»;

– бідна реалізація контексту.

1.2.2 FluentValidation

1.2.1.1 Загальний опис

FluentValidation – це бібліотека валідації у середовищі .NET із відкритим кодом. Вона надає Fluent API, і використовує лямбда-вирази для створення правил перевірки. Поточна стабільна версія (8.0.101) підтримує як .NET Framework 4.5+, так і .NET Standard 1.0. Крім того, вона має інтеграцію для ASP.NET MVC 5, ASP.NET Web API та ASP.NET Core MVC.

Кореневим елементом бібліотеки є інтерфейс IValidator<in T>, із методом Validate(T), який повертає ValidationResult – сукупність повідомлень про завалені правила перевірки та властивість IsValid. Даний інтерфейс імплементується абстрактним класом AbstractValidator<T>, що надає засоби для написання конкретних класів-валідаторів і зберігання предикатів перевірки бізес-правил.

FluentValidation постачається з декількома вбудованими валідаторами. Повідомлення про помилку для кожного валідатора може містити спеціальні заповнювачі, які будуть заповнені при побудові повідомлення про помилку.

Таблиця 1.3 – Вбудовані валідаційні правила

Функція	Призначення
NotNull()	Валідує, що зазначена властивість не є нульовою
NotEqual(T) / (Action <T1>)	Валідує, що значення вказаного ресурсу не дорівнює певному значенню (або не дорівнює значенню іншого властивості)
NotEmpty()	Валідує, що зазначена властивість не є нульовою, порожня рядок чи пробіл (або значення за замовчуванням для типів значень, наприклад 0 для int)
Equal(T) / (Action <T1>)	Валідує, що значення вказаного властивості дорівнює певному значенню (або рівному значенню іншого властивості)
Length(int, int)	Валідує, що довжина певного властивості рядка знаходиться в межах зазначеного діапазону, однак, ігнорує null-значення.
MaximumLength (int)	Валідує, що довжина рядка певної властивості не перевищує вказане значення.
MinimumLength(int)	Валідує, що довжина певного властивості рядка перевищує вказане значення.
LessThan(T) / (Action <T1>)	Валідує, що значення вказаного ресурсу менше, ніж певне значення (або менше, ніж значення іншого властивості)
LessThanOrEqua lTo(T)/(Action <T1>)	Валідує, що значення вказаного ресурсу менш або дорівнює певному значенню (або значенню іншого ресурсу)
GreaterThan(T) / (Action <T1>)	Валідує, що значення вказаного властивості перевищує певне значення (або перевищує значення іншого властивості)
GreaterThanOrE qualTo(T) / (Action<T1>)	Валідує, що значення вказаного властивості більше або дорівнює певному значенню (або більше або дорівнює значенню іншого властивості)
Matches(regex)	Валідує, що значення вказаного властивості відповідає даному регулярному виразу

Продовження таблиці 1.3.

Функція	Призначення
EmailAddress()	Валідує, що значення вказаного ресурсу є дійсним форматом адреси електронної пошти

Окрім готових валідаторів, бібліотека надає можливість доповнення їх набору власними. Існує кілька способів створити індивідуальний багаторазовий валідатор. Рекомендований спосіб полягає у використанні Predicate Validator для написання користувальницької перевірки, але також можна написати власну реалізацію класу PropertyValidator. Валідатор-предикат (Must(Func<T, bool>)) передає значення вказаної властивості в делегат, який може виконувати користувальницьку логіку перевірки на відповідність інваріантам.

1.2.2.2 Переваги

Ключовими відмінністями FluentValidation серед решти валідаційних засобів є легкість тестування та можливість легко задавати правила перевірки однієї властивості відносно іншої. Наприклад, нехай є поле «field», значення якого потрібно перевіряти, якщо «field2» встановлено в true. Подібну перевірку можна описати одним рядком коду: «RuleFor(x => x.field).NotEmpty().When(x => x.field2);».

Дана бібліотека поставляється з двома методами розширення: Should Have Validation Error For, Should Not Have Validation Error For – це значно полегшує написання модульних тестів для валідаторів. Також ці методи мають підтримку тестового фреймворку NUnit за змовчуванням. Метод ShouldHaveChildValidator дає можливість перевірити, що складна властивість має власний вкладений валідатор. Інший варіант полягає у використанні сторонньої бібліотеки, такої як FluentValidation.Validators.UnitTestExtension. Це забезпечує альтернативний синтаксис для перевірки валідаторів.

Клас-валідатор легко підключити до моделі, що перевіряється. Робиться це шляхом додавання атрибута FluentValidation.Attributes. ValidatorAttribute. Для інтеграції із MVC фреймворком достатньо лише викликати метод

FluentValidationModelValidatorProvider .Configure() у Application_Start. Таким чином можна повністю замінити System. ComponentModel.DataAnnotations.

Загалом, можна виділити такі переваги:

– зручний для читання та написання Fluent API;

– декларативний синтаксис;

– умовне підтвердження різних властивостей набагато простіше порівняно з анотаціями даних;

– широкий вибір імплементованих валідаторів;

– можливість розширення готового набору валідаторів шляхом написання власного класу, що наслідує AbstractValidator;

– можливість реалізації «Always valid entity» шляхом впровадження класу-валідатора як залежності у саму сутність;

– можливість реалізації Notification патерну;

– можливість реалізації Specification патерну шляхом виділення окремого класу-валідатора під кожне правило-специфікацію;

– можна легко задавати перевірки для моделі практично будь-якої складності;

– набагато кращий контроль правил перевірки;

– відокремлення логіки валідації від моделі;

– написання юніт-тестів набагато простіше порівняно з анотаціями даних;

– відмінна підтримка перевірки на стороні клієнта для більшості стандартних правил перевірки.

1.2.2.3 Недоліки

Єдиним недоліком, який можна виділити є неповноцінна реалізація контексту валідації. У якості контексту можна передавати довільні дані в конфігураційний конвеєр RootContextData (Dictionary<string, object>), який можна отримати в межах власних валідаторів властивостей. Розробники рекомендують прибігати до цього засобу, якщо потрібно зробити умовне рішення, засноване на

довільних даних, недоступних в об'єкті, який перевіряється, оскільки валідатори не мають стан.

Більш детально недоліки даної реалізації контексту можна представити наступними твердженнями:

– відсутність безпеки типів;

– єдиний код працює із будь-яким контекстом, що призводить до розростання класів-валідаторів;

– складність покриття юніт-тестами валідації, що пов’язана із виконанням операцій у контексті.

1.3 Висновок до розділу

У даному розділі було описано теоретичні засади валідації (зокрема, контекстуальної) з точки зору дизайну, рекомендовані практики організації коду логіки перевірки інваріантів, а також розглянуто найпопулярніші існуючі рішення, їх переваги та недоліки. Проаналізована інформація лягла в основу розробки бібліотеки контекстуальної валідації.

У реальних системах існує складна логіка, яка вирішує, коли суб'єкт є дійсним для конкретних операцій, а коли ні. Проблема полягає в тому, що модель визначає єдиний набір перевірок, однак вона повинна поглинати різні набори даних за різних обставин. Існує рідко тільки один контекст для того, щоб бути дійсним чи недійсним. Набагато корисніше розглядати валідаційне правило як те, що пов'язано з контекстом. Як правило, дії, котрі потребують попередньої валідації сутності одного типу, можуть виконуватися у різних класах, за різних обставин, тобто у різних контекстах. Так постає проблема контекстуальних перевірок. Цей тип валідації є основною частиною бізнес-логіки, що використовується для споживання та перевірки вводу користувача або інших акторів у систему, і повинна бути виділена в окремі сутності.

Не зважаючи на поширеність проблеми, популярні фреймворки надають бідний інтерфейс для реалізації контекстуальної валідації.

Так, DataAnnotations методи TryValidateObject, ValidateObject та ValidateProperty статичного класу Validator, а також метод Validate інтерфейсу IValidatableObject споживають ValidationContext. Цей клас описує тип або член, на якому виконується перевірка. Він також дозволяє додавати користувальницьку перевірку за допомогою будь-якої служби, яка реалізує інтерфейс IServiceProvider. Це і дає можливість розширення валідації контекстом, проте у дуже не зручний спосіб. Більше того ValidationContext є огорткою для Object типу (тобто будь-якого), що негативно позначається на безпеці типів програми.

FluentValidation також не реалізує підтримку контексту. Єдине, чим виражається контекст у ході валідації – це Dictionary<string,object> (хеш-таблиця зі строковим ключем). Джеремі Скіннер (голова розробників даного фреймворку) особисто зазначив, що додавання прямої підтримки для деякого роду контексту виходить за межі задачі проекту [12].

Решта існуючих фреймворків та бібліотек вирішують описану проблему ще гірше, або взагалі її ігнорують. Саме тому постає потреба у створенні нового засобу для вирішення проблеми контекстуальної валідації – бібліотеки, яка описується у наступних розділах.

2 ФОРМУВАННЯ ВИМОГ ДО БІБЛІОТЕКИ

Отже, існує потреба у створенні нового фреймворку валідації у .NET середовищі, що ставить у центр саме контекстуальну валідацію.

На основі базових засад валідації, популярних практик у світі .NET, а також аналізу існуючих до фінального продукту висуваються такі вимоги:

- сприяти зменшенню кількості коду, необхідного для реалізації валідаційної логіки;
- мати можливість використання у підходах відкладеної валідації [2] та завжди валідної сутності [3];
- мати інтуїтивно зрозумілий API;
- ураховувати нативні правила валідації .NET (System.ComponentModel.DataAnnotations);
- створювати правила за Exception та Notification підходами [7], оскільки вони базові та загальноприйняті;
- сприяти зменшенню цикломатичної складності;
- сприяти зменшенню Design Smells (Rigidity, Fragility, Immobility, Viscosity та Needless Complexity), Code Smells, Coupling, збільшенню Coherency [11], отже створювати позитивний вплив на дизайн застосунку;
- гарантувати безпеку типів при контекстуальній валідації;
- спрощення написання юніт-тестів;
- асинхронна валидація;
- реалізація продукційних правил у вигляді шаблону Specification;
- інтеграція з ASP.NET та ASP.NET Core.

3 ОПИС БІБЛІОТЕКИ КОНТЕКСТУАЛЬНОЇ ВАЛІДАЦІЇ

Бібліотека контекстуальної валідації у даній роботі являє собою елемент науково-практичної новизни. Даний розділ присвячено опису самої бібліотеки: типів, що її складають, її можливостей, переваг та недоліків.

3.1 Загальний опис

Розроблена бібліотека контекстуальної валідації – це система продукційних правил (Production Rule System [13]), що організує логіку перевірок за допомогою набору правил, кожне з яких має умову (предикат Func<TEntity, TContext, bool>) та делегат реакції на її порушення, що задається функціями NotifiedAs та ThrowingException.

Для конструювання правил використовується Fluent API – різновид Internal Domain Specific Language (підможина мови програмування, що орієнтована на певний тип проблеми). API розроблено за рекомендаціями Фаулера [14] та із використанням патернів Closure / Nested Closure, ExpressionBuilder, Function Sequence та Method Chaining для побудови правил і Notification для організації результату валідації (у т.ч. для самовалідації стану бібліотеки). Даний підхід забезпечує легке читання та написання логіки правил перевірки. Приклади коду наведені у Додатку А.

Продукт підтримує два підходи до валідацій, відомі як Notification та Exception Approach. Перший реалізується через інтерфейс INotificationValidationRule, що у результаті валідації повертає ValidationResult – агрегат перерахування структур Notification та загального валідаційного повідомлення. Останній підхід, що реалізується через інтерфейс IExceptionValidationRule – викидає користувацьке виключення при порушенні валідаційного правила.

Бібліотека працює у двох режимах: загальноприйнятний підхід при якому правила інкапсулюються в окремих об’єктах – динамічна валідація, та підхід для швидкої розробки – статична валідація.

Перший використовує Factory та Builder [15] для генерації валідаційних правил. Даний підхід надає можливість впровадження валідаційних правил за допомогою механізму Dependency Injection. Таким чином правила можуть бути застосовані для відкладеної валідації [2] та завжди валідної сутності [3]. Також виокремлення правил в окремі класи полегшує тестування та сприяє Low Coupling / High Coherency, що у свою чергу вирішує проблеми Design та Code Smells.

Останній режим використовує інкапсулювання усіх валідаційних предикатів у статичному класі DomainModelValidator із публічними методами Configure та Validate. Таким чином можна викликати валідацію, наприклад, в GenericRepository [16] перед записом у сховище і вона відбуватиметься для всіх сутностей, помічених атрибутом ValidateDomainConstraintsAttribute. Цей підхід значно зменшує час розробки ПЗ, проте має негативний вплив на дизайн додатку. Його використання є виправданим для умов обмеженого часу або написання програм, що не матимуть подальшого розвитку. Тому бібліотека може працювати у «перехідному» режимі: фабрика має здатність конструювати правила зі статичного конфігу. Таким чином забезпечується безшовний перехід із другого режиму до першого.

Бібліотека підтримує нативну валідацію .NET за змовчуванням, яку можна вимкнути за допомогою методів DisableNetNativeValidation при статичному конфігуруванні або IgnoringNetNativeValidation у відповідного Builder при конструюванні через Factory.

3.2 Опис абстракцій бібліотеки

Для зменшення зв’язанності клієнтського коду та поліпшення тестування користувачеві бібліотеки пропонується працювати не з конкретними класами, а з абстракціями, що акомпонує принципу Dependency Inversion. Нижче описуються інтерфейсні та абстрактні типи, які являють собою контракт бібліотеки контекстуальної валідації. Усі вони знаходяться у просторі імен DomainModelValidation.Abstract.

Таблиця 3.1 – Опис інтерфейс них типів контракту

Методи	Значення
IDomainModelValidatorConfigurator<TContext> – представляє набір правил, поділених на види операцій	
ConfigureOnUpdate	Налаштувати доменні обмеження для перевірки перед виконанням дії UPDATE
ConfigureOnCreate	Налаштувати доменні обмеження для перевірки перед виконанням дії CREATE
ConfigureOnDelete	Налаштувати доменні обмеження для перевірки перед виконанням дії DELETE
DisableNetNativeValidation	Відключити урахування валідації System. ComponentModel. DataAnnotations
IExceptionRule<TEntity, TContext> – представляє собою набір предикатів валідації, які об'єднуються та викликаються в порядку додавання. У випадку порушення правил буде викинуте виключення. Одне правило для кожного типу в сукупності	
Including	Об'єднати з попереднім правилом перевірки
ThrowingException	Визначає виняток, який буде викинуто в разі порушення правил перевірки. Виняток за замовчуванням, якщо не встановлено
IExceptionRuleBuilder<TEntity, TContext> : ISimpleExceptionRuleBuilder <TEntity, TContext> - контракт паттерну Builder для побудови валідаційних правил, що викидають виключення	
IgnoringNetNativeValidation	Вимкнути підтримку врахування валідації System. ComponentModel. DataAnnotations для правила, що створюється
FromConfig	Побудувати копію правила зі статичного конфігу
IExceptionRuleSet<TContext> - представляє набір правил, що викидають виключення	

Продовження таблиці 3.1.

Методи	Значення
AddRule	Додати валідаційне правило передавши предекат
IExceptionValidationRule<TEntity, TContext> – представляє собою валідаційне правило, що викидає виключення	
Validate	Провалідувати. Викинути виключення у разі порушення інваріанту
INotificationRule<TEntity, TContext> : IValidationRule<TEntity, TContext> – представляє правило, що повертає список повідомлень про результат валідації	
Including	Об'єднати з попереднім правилом перевірки
NotifiedAs	Визначити повідомлення про порушення попереднього правила
WithValidationExceptionMessage	Визначити повідомлення Domain Model Constraint Violation Exception для цього правила
INotificationRuleBuilder<TEntity, TContext> : ISimpleNotificationRuleBuilder <TEntity, TContext> – контракт паттерну Builder для побудови валідаційних правил, що повертають список повідомлень	
IgnoringNetNativeValidation	Вимкнути підтримку врахування валідації System. ComponentModel. DataAnnotations для правила, що створюється
FromConfig	Побудувати копію правила зі статичного конфігу
INotificationRuleSet<TContext> – представляє набір правил, що повертають список повідомлень	
AddRule	Додати валідаційне правило передавши предекат
INotificationValidationRule – представляє собою валідаційне правило, що повертає список повідомлень	
Validate	Здійснити валідацію та повернути ValidationResult
IRuleFactory – контракт патерну AbstractFactory, що створює різні правила перевірки	

Продовження таблиці 3.1.

Методи	Значення
CreateExceptionRule	Створити правило, що викидає виключення, на основі делегату білдера
CreateNotificationRule	Створити правило, що повертає ValidationResult, на основі делегату білдера
ISimpleExceptionRuleBuilder<TEntity, TContext> – базовий інтерфейс для побудови правил, що викидають виключення	
BuildRule	Будує правило на основі валідаційного предикату
ISimpleNotificationRuleBuilder<TEntity, TContext> – базовий інтерфейс для побудови правил, що повертають ValidationResult	
BuildRule	Будує правило на основі валідаційного предикату
IValidationRuleSet<TContext> – базовий інтерфейс набору правил валіації	
UsingNotificationApproach	Повертає набор правил, що повертають повідомлення
UsingExceptionApproach	Повертає набор правил, що викидають виключення

Окрім інтерфейс них типів, простір імен DomainModelValidation. Abstract також містить єдиний абстрактний клас Rule<TEntity, TContext>, котрий є базовим типом, що представляє будь-якого виду валідаційні правила. Даний тип не входить у контракт бібліотеки, проте імплементує інтерфейс IValidationRule<TEntity, TContext>.

У загальному випадку стан валідаційного правила зберігається у наступних полях та властивостях.

Таблиця 3.2 – Опис стану абстрактного класу Rule

Поле або властивість	Тип	Зміст	Примітка
_validationRulePredicates	List<Func<TDbEntity, TUnitOfWork, bool>>	Список валідаційних предикатів	protected

Продовження таблиці 3.2.

Поле або властивість	Тип	Зміст	Примітка
IsNetNativeValidation Enabled	bool	Це поле визначає, чи повинні застосовуватися правила валідації System. ComponentModel. DataAnnotations на данному правилі	public{ get; internal set; }
_isEntityValid	bool	Визначає валідність сутності у даній валідації	[ThreadStatic], protected, static
_basicException	DomainModel ConstraintViol ationException	Виключення, шо викидається у разі порушення правила перевірки	[ThreadStatic], protected, static

Варто зауважити, що виключення _basicException викидається лише у випадку використання статичного валідатора по сутності, котра помічена атрибутом ValidateDomainConstraintsAttribute. Даний підхід не рекомендується використовувати для розробки довготривалих проектів, над якими працює декілька команд або велика кількість людей. Проте для короткотривалих проектів зі стиснутими термінами та малою кількістю розробників даний підхід є рекомендованим.

Робота з абстрактним класом Rule у даному випадку схожа на логіку використання типу AbstractValidator із бібліотеки FluentValidation. Так, задача конкретних типів виключень полягає у тому, щоб перевизначити абстрактні члени, які задають поведінку базового класу валідаційного правила. Нижче подано опис методів даного класу.

Таблиця 3.3 – Опис поведінки абстрактного класу Rule

Методи / властивості	Значення
ValidateRule	Слугує каркасом загального випадку валідації. Використовує патерн Шаблонний Метод, викликаючи метод ApplyNetNativeValidationRules та абстрактний метод DefineEntityValidity
BasicIncluding	Додає новий предикат до списку валідаційних предикатів
DefineEntityValidity	Абстрактний метод, що має визначати валідність сутності.
ApplyNetNativeValidationRules	Валідує сутність по правилам System. ComponentModel.DataAnnotations
ValidationPredicatesClone	Клон списку валідаційних предикатів

У випадку статичної валідації використовується метод ValidateRule, котрий приймає сутність та об’єкт контексту. У разі порушення хоча б одного з валідаційних предикатів метод викидає виключення незалежно від конкретного типу валідаційного правила. У випадку використання нестатичних валідаторів, даний метод не використовується. Він викликається лише всередині статичного класу DomainModelValidator<TContext>.

3.3 Опис користувацький виключень

Для кращої інтеграції із клієнтським кодом введено набір типів користувацьких виключень, кожен із яких унаслідується напряму від системного типу System.Exception. Таким чином, код, що використовує бібліотеку може найбільш точним способ відреагувати на неправильно конфігурацію, передачу параметрів тощо, шляхом перехвату необхідного класу виключення у блоках try-catch.

Таблиця 3.4 – Опис користувацьких виключень

Тип	Значення
DomainModelConstraintViolationException	Викидається у загальному випадку статичної валідації. Містить перелік повідомлень про порушені інваріанти та загальне повідомлення.
DomainModelValidatorConfigurationException	Викидається у ході неправильної конфігурації статичного валідатора.
RulesSetNotSpecifiedException	Викидається у випадку валідації сутності по контексту, для якого не сконфігуровано набір валідаційних правил (у разі статичної валідації).
ValidationRuleUnregisteredException	Викидається у разі валідації по сконфігурованому контексту, проте не сутності.

Дані виключеня використовуються у випадку статичної валідації, у решті – використовуються стандартні виключеня платформи .NET.

3.4 Опис основних типів

Як було зазначено вище, бібліотека контекстуальної валідації працює у двох режимах: динамічної та статичної валідації. Нижче описано типи, котрі використовуються для статичної валідації та їх методів, полів, змінних, які не є реалізаціями тих чи інших інтерфейсів або ж є такими, проте потребують додаткового роз’яснення принципу їх роботи.

Таблиця 3.5 – Опис типів, що використовуються виключно для статичної валідації

Елементи типу	Значення
DomainModelValidator<TContext> – статичний валідатор. Містить набір правил, поділених на види операцій (onUpdate, onCreate, onDelete).	

Продовження таблиці 3.5.

Validate	Валідує сутність TEntity відносно контексту типу TContext шляхом виклику методу ValidateRule у відповідному об’єкті RulesSet
Configure	Конфігурує набір валідаційних правил за допомогою делегата, що приймає IDomainModelValidatorConfigurator
DomainModelValidatorConfigurator<TContext> – імплементація інтерфейсу IDomainModelValidatorConfigurator<TContext>, через методи якого встановлюється стан об’єкту: OnUpdateRulesSet, OnDeleteRulesSet, OnCreateRulesSet та IsNetNativeValidationEnabled	
RulesSet – імплементує інтерфейси IValidationRuleSet, IExceptionRuleSet та INotificationRuleSet	
Dictionary<Type, IValidationRule< object, TUnitOfWork>> _rules	Словник із валідаційними правилами. Ключем виступає тип сутності, що валідується
CheckAttribute	Викликається всередені AddRule для перевірки наявності атрибуту ValidateDomainConstraints Attribute у метаданих сутності, що валідується. Викидає користувацьке виключення у разі його відсутності.
ValidateDomain ConstraintsAttrib ute	Атрибут, яким необхідно помічати сутності, котрі підлягають статичній валідації
ValidationType	Перерахування типів статичної валідації (OnUpdate, OnCreate, OnDelete). Рекомендовано використовувати із поєднанням патерну GenericRepository<T> для розділення наборів правил по типам операцій над даними

Таблиця 3.6 – Опис типів, що використовуються для статичної та динамічної валідації

Елементи типу	Значення
ExceptionRule<TDbEntity, TUnitOfWork> – наслідує клас Rule та імплементує інтерфейси IExceptionRule та IExceptionValidationRule	
Including	Викликає всередині себе BasicIncluding, передаючи валідаційний предикат
ThrowingException	Викликає всередині себе BasicIncluding, передаючи предикат, котрий викидає виключення у разі !_isEntityValid
Clone	Повертає новий екземпляр ExceptionRule, який є клоном даного об’єкту
Validate	Явна реалізація інтерфейсу IExceptionValidationRule. Викликає всередині себе ValidateRule базового класу. Викидає виключення у випадку порушення інваріантів. Використовується виключно у динамічній валідації
DefineEntityValidity	Реалізація абстрактоного методу. Повертає значення _isEntityValid
Notification – слугує контейнером даних результату перевірки, елемент реалізації Notification патерну	
string ErrorMessage	Повідомлення про помилку
string MemberName	Член типу, який порушив інваріанту
Type InstanceType	Тип, представник якого порушив інваріанту
NotificationRule <TDbEntity, TUnitOfWork> – наслідує клас Rule та імплементує інтерфейси INotificationRule та INotificationValidationRule	
Including	Викликає всередині себе BasicIncluding, передаючи валідаційний предикат

Продовження таблиці 3.6.

Елементи типу	Значення
NotifiedAs	Викликає всередині себе BasicIncluding, передаючи валідаційний предикат, котрий додає повідомлення до списку повідомлень _basicException у разі !_isEntityValid
Clone	Повертає новий екземпляр NotificationRule, який є клоном даного об’єкту
Validate	Явна реалізація інтерфейсу INotificationValidationRule. Викликає всередині себе ApplyNetNativeValidationRules. Вираховує результат валідації на основі стану _basicException. Використовується виключно у динамічній валідації
DefineEntityValidity	Реалізація абстрактоного методу. Повертає значення істину у разі істинності _isEntityValid та відсутності повідомлень у _basicException
ClearRuleState	Зачищує список повідомлень після кожно валідації.
ValidationResult – представляє собою результат валідації	
IEnumerable<Notification> Notifications	Перерахування повідомлень про порушення інваріантів
ValidationMessage	Загальне повідомлення
IsValid	Індикатор успішності проходження валідації

Таблиця 3.7 – Опис типів, що використовуються виключно для динамічної валідації

Елементи типу	Значення
ExceptionRuleBuilder – імплементація інтерфейсу IExceptionRuleBuilder	

Продовження таблиці 3.7.

ExceptionRule<TDbEntity, TUnitOfWork> _rule	Правило, що будується
bool isNetNativeValidationEnabled	Передається у конструктор ExceptionRule для визначення врахування правил валідації System. ComponentModel.DataAnnotations
NotificationRuleBuilder – імплементація інтерфейсу INotificationRuleBuilder	
NotificationRule<TDbEntity, TUnitOfWork> _rule	Правило, що будується
bool isNetNativeValidationEnabled	Передається у конструктор ExceptionRule для визначення врахування правил валідації System. ComponentModel.DataAnnotations
RuleFactory– імплементація інтерфейсу IRuleFactory	
CreateExceptionRule	Повертає ExceptionRule, що сконструйовано за допомогою делегату, котрий приймає IExceptionRuleBuilder
CreateNotificationRule	Повертає NotificationRule, що сконструйовано за допомогою делегату, котрий приймає INotificationRuleBuilder

3.5 Опис зовнішніх валідаторів

Наразі перелік зовнішніх валідаторів обмежується лише нативною валідацією середовища .NET – System.ComponentModel.DataAnnotations. Дана логіка зосереджується у класі NetValidator із єдиним статичним методом Validate, котрий валідує сутність без урахування контексту та повертає перерахування повідомлень IEnumerable<Notification>, які являють собою власне результат валідації.

Всередені зовнішнього валідатору використовується статичний метод Validator.TryValidateObject із простору імен System. ComponentModel. DataAnnotations, а оригінальний результат валідації відображається на перерахування Notification із бібліотеки.

У планах на подальший розвиток бібліотеки реалізація зовнішнього валідатору, котрий би працював із правилами, що задаються за допомогою FluentValidation, а також розширення існуючої логіки: робота із ValidationContext.

3.6 Техніки, що застосовувались у ході розробки бібліотеки

У ході розробки використовувалися різноманітні стандартні та рекомендовані техніки і прийоми розробки як спеціалізованих бібліотек чи фреймворків, так і C# коду вцілому. Нижче описані деяки з них.

3.6.1 Техніки розробки DSL

Спеціальні мови домену (Domain Specific Languages, DSL) [14] – це невеликі мови, зосереджені на певному аспекті програмної системи. DSL часто використовуються в системі, головним чином написаною загальноприйнятою мовою.

DSLs мають дві основні форми: зовнішні та внутрішні. Зовнішній DSL – це мова, яка парситься незалежно від загальної мови хосту: хороші приклади включають регулярні вирази та CSS. Зовнішні DSL мають сильні традиції в спільноті Unix. Внутрішні DSL – це особлива форма API в загальноприйнятій мові, яка часто називається вільним (fluent) інтерфейсом. Приклади DSL – LINQ to SQL, LINQ to XML, EF Fluent API.

Цінність DSL полягає у тому, що добре спроектований DSL може бути набагато простішим у використанні, ніж традиційна бібліотека. Це покращує продуктивність програміста, що завжди є цінним. Зокрема, це також може покращити спілкування з доменними експертами, що є важливим інструментом для вирішення однієї з найважчих проблем розробки програмного забезпечення – вирішення складності в серці програмного забезпечення.

Специфічна мова домену (DSL) – це мова програмування з обмеженою виразністю, орієнтована на певний домен. Кожен DSL може обробляти лише один конкретний аспект системи. DSLs широко використовуються в колах Unix розробників із перших днів роботи цієї системи. Більшість IT-проектів використовують декілька DSL: CSS, SQL, регулярні вирази тощо.

Зовнішній DSL – це зовсім окрема мова, яка розбирається в даних, які мова мови гостя може зрозуміти. Внутрішні DSL просто використовують звичайні засоби мови програмування, які ви використовуєте в будь-якому випадку.

Внутрішній DSL – це лише особлива форма написання коду на хостовій мові загального призначення. Як такі вони часто називаються "вільні" інтерфейси або вбудовані DSL.

Саме внутрішній DSL використовується для реалізації контракту бібліотеки.

Найцікавішою перевагою, є те, що добре продуманий внутрішній DSL може бути зрозумілим для стейкхолдерів із бізнес сторони, дозволяючи їм безпосередньо осмислити код, який реалізує бізнес-правила. Тоді вони зможуть переглянути їх для точності, поговорити про них з розробниками та внести проектні зміни, щоб розробники могли правильно їх реалізувати. Отримання DSL-систем для читання бізнесу є набагато меншим, ніж бізнес, який можна записувати, але дає більшу частину переваг.

Наразі вже існує безліч різноманітних фреймворків, які програмісти повинні навчитися. Це неминучий наслідок багаторазового використання програмного забезпечення, що є єдиним способом, яким ми можемо отримати повну інформацію про все те, що програмне забезпечення має робити в ці дні. По суті, DSL – фасад над фреймворком. В результаті вони створюють невелику складність над тим, що вже є. Дійсно, хороший DSL повинен поліпшити ситуацію, полегшуючи використання цих систем.

У ході розробки бібліотеки контекстуальної валідації використовувалась низка описаних Мартіном Фаулером та Ребеккою Парсонс патернів DSL із книги «Спецефічні Мови Домену».

Таблиця 3.8 – Застосування патернів DSL

Назва патерну	Опис	Місця застосування у бібліотеці
Аннотація	Дані про елементи програми, такі як класи та методи, які можна обробити під час складання чи виконання	Атрибут ValidateDomain ConstraintsAttribute
Замикання	Блок коду, який може бути представлений як об'єкт (або структура даних першого класу), і нерозривно розміщений у потоці коду, дозволяючи йому посилатися на його лексичну сферу	Замикання валідаційних предикатів на полі isValid.
Білдер конструкцій	Поступове створення незмінного об'єкту за допомогою білдера, який зберігає аргументи конструктора в полях	IExceptionRuleBuilder, INotificationRuleBuilder
Білдер виразів	Об'єкт або сімейство об'єктів, що забезпечує безперебійний інтерфейс за допомогою звичайного API-командного запиту	Використання типів білдерів у Action делегатах
Послідовність функцій	Комбінація функцій викликів як послідовність висловлювань	Вираження валідаційних предикатів у вигляді списку делегатів Func<TEntity, TContext, bool>

Продовження таблиці 3.8.

Зчеплення методів	За допомогою методів модифікатора повертає об'єкт-хост, так що декілька модифікаторів можна викликати в одному виразі.	IExceptionRule, INotificationRule при виклику Including, ThrowingException, NotifiedAs
Вкладені замикання та вкладені функції	Підключення виклику функції, ввівши їх у замикання в аргументі	Передача делегату як параметра делегату
Повідомлення	Збирати помилки та інші повідомлення для клієнтського коду.	INotificationRule, INotificationRuleBuilder, Notification
Система продукційних правил	Організація логіки за допомогою набору продукційних правил, кожне із яких має умову та дії	Уся бібліотека вцілому побудована на даному прийомі

3.6.2 Прийом EIMI

Прийом явної реалізації інтерфейсного методу (EIMI, Explicit Interface Method Implementation) використовується, якщо клас реалізує два інтерфейси, що містять члени з однаковою сигнатурою, тоді реалізація цих членів в класі призведе до того, що обидва інтерфейси використовуватимуть ці елементи як їх реалізацію. Проте, якщо члени інтерфейсу не виконують однакову функцію, це може призвести до неправильної реалізації одного або обох інтерфейсів. Можна реалізувати елемент інтерфейсу, який явно створює член класу, який викликається тільки через інтерфейс, і є специфічним для цього інтерфейсу. Це досягається шляхом найменування члена класу з назвою інтерфейсу та крапки.

EIMI-метод не може бути віртуальним, а значить, його не можна перевизначити. Це відбувається тому, що EIMI-метод насправді не є частиною

об'єктної моделі типу; це всього лише засіб зв’язування інтерфейсу (набору варіантів поведінки, або методів) із типом [17].

У бібліотеці контекстуальної валідації прийом EIMI використовується у конкретних типах правил: NotificationRule та ExceptionRule для розмеження статичної (безінтерфейсна реалізація методу Validate) та динамічної (явна імплементація INotificationValidationRule та IExceptionValidationRule відповідно) валідації.

3.6.3 Патерни GoF

Значна частина коду, який відповідає за динамічну валідацію, організована за допомогою патернів Банди Чотирьох [15]. Нижче наведено опис типів реалізованих у вигляді дизайн-шаблонів.

Таблиця 3.9 – Застосування патернів GoF

Назва патерну	Опис	Місця застосування у бібліотеці
Абстрактна фабрика	Інтерфейс для створення сімейств взаємопов'язаних об'єктів з певними інтерфейсами без вказівки конкретних типів даних об'єктів. Патерн реалізовано не повністю	Інтерфейс IRuleFactory та клас RuleFactory. Сімейство об’єктів-продуктів – IexceptionValidationRule та INotificationValidationRule
Будівник	Інкапсулює створення об'єкту та дозволяє розділити його на різні етапи	Інтерфейси ISimple Notification Rule Builder, ISimple Exception Rule Builder, INotification Rule Builder, IException Rule Builder та їх реалізації Exception Rule Builder і Notification Rule Builder

3.7 Переваги використання бібліотеки

Насамперед, переваги від використання даної бібліотеки контекстуальної валідації – як і в існуючих рішеннях – проявляються у позитивному впливі на дизайн додатку: зменшення зв’язаності та підвищення зв’язності, сприяння організації коду згідно із принципами SOLID та метапринципами DRY [18] і KISS [18], покращення показників таких метрик коду, як цикломатична складність.

3.7.1 Зменшення об’єму коду валідації

Якщо аналізувати вплив бібліотеки контекстуальної валідації на дизайн застосунку в широкому сенсі, її використання сприяє організації коду за DRY, KISS та YAGNI – трьом основним метапринципам дизайну, які, наряду з іншими, дозволяють регулювати складність розробки.

Принцип KISS (Keep It Simply Stupid) полягає у тому, що найпростіше рішення щодо зменшення складності – це розділити систему на дрібні, незалежні модулі, якими простіше управляти, при цьому кожна частина даних повинна мати чітке, надійне представлення в системі – типово, класс.

Принци DRY (Don’t Repeat Yourself) полягає у тому, що вищеописані артефакти ПЗ – модулі, класи, інтерфейси, структури, функції тощо – зустрічаються у програмній системі лише один раз. Таке рішення проблеми з урахуванням мінімальної кількості залежностей можна пояснити принципом «бритви Оккама», який знаходить застосування не лише у галузі розробки програмного забезпечення.

Винесення коду перевірки дійсності інваріантів сприяє метапринципу DRY, адже класи-валідатори можуть бути перевикористані у різних типах. Також внаслідок цього реалізується метапринцип KISS, оскільки клас, що виконує дії над сутностями бере на себе менше відповідальностей. Таким чином, це призводить до зменшення коду валідаційної логіки.

3.7.2 Використання у підходах відкладеної валідації та завжди валідної сутності

За допомогою даної бібліотеки легко реалізувати два основних підходи до організації логіки перевірки інваріантів. За рахунок того, що з використанням фабрик правил у динамічному підході логіка валідації відокремлюється від решти коду типу, об’єкти класів-валідаторів можна впроваджувати як залежності або ж всередину самих типів та здійснювати перевірки перед кожною зміною стану, або ж всередину, наприклад, сервісів і здійснювати валідацію за способом Дж. Палермо.

3.7.3 Інтуїтивно зрозумілий API

Можливість легкого читання коду – одна з головних проблем дизайну. Ще у 1984 професор Дональд Кнут – один із батьків-засновників комп'ютерних наук – опублікував книгу «Літературне програмування», в якій підкреслив ідею, що програма повинна бути написана як есе. Так, сучасні мови та засоби програмування розроблені, радше, для людей, а не комп'ютера і радикально відрізняються від байт-коду, у який вони в наслідку компілюються.

Неминуча правда розробки програмного забезпечення полягає в тому, що код постійно змінюється. Вартість цієї зміни завжди збільшується, коли намір коду недостатньо зрозумілий. Тому існує потреба застосовувати методи, які полегшують читання та розуміння коду, оскільки ціна рефакторингу може бути астрономічною.

Fluent API – це потужні та зручні абстракції на готовому рівні з кількома натисканнями клавіш, що підкреслюють мету вашого коду, подають її у вигляді, натурально, речення. Приклад використання Fluent API із Додатку Б можна прочитати як речення: «Фабрика, створи правило, що викидає виключення для сутності типу Entity та контексту типу Context, застосувавши білдер. Білдер, збудуй правило із предикатом P1, що викидає виключення ArgumentException, включаючи предикат P2, що викидає виключення Exception».

Таким чином, інтуїтивно зрозумілий інтерфейс є значною перевагою, адже забезпечує легкість написання та підтримування валідаційних правил.

3.7.4 Робота з System.ComponentModel.DataAnnotations

За змовчуванням, бібліотека контекстуальної валідації підтримує роботу з нативною валідацією .NET – правилами переіврки, що задаються за допомогою атрибутів із простору імен System. ComponentModel. DataAnnotations. При цьому надається можливість вмикати та вимикати дану властивість як для окремих правил при їх конструюванні, так і вцілому при роботі зі статичною конфігурацією.

3.7.5 Exception та Notification підходи

Два основних способи організації реакцій на порушені предикати інваріантів: викидання виключень та повертання повідомлень реалізуються у випдках як статичної (за допомогою Rule.Validate), так і динамічної (за допомогою IExceptionValidationRule, INotificationValidationRule тощо) валідації. При використанні зовнішніх валідаторів відповідь на провалену валідацію також здійснюється в одному зі сконфігурованих форматів.

3.7.6 Зменшення цикломатичної складності

Написання інваріантів за допомогою Fluent API, а не тривіальних if-else конструкцій сприяє зменшенню цикломатичної складності як клієнтського коду, так і коду власне валідаційних правил, оскільки зводить нанівець кількість розгалужень (хоча самі розгалуження реалізуються всередині бібліотекою контекстуальної валідації).

3.7.7 Подолання Code Smells

Внаслідок того, що контекстуальна валідація не реалізується у повній мірі наявними спеціалізованими бібліотеками та фреймворками, код, що її імплементує зосереджується прямо у класах, що виконують дії, які потребують

попередньої перевірки дотримання інваріантів: наприклад, у класах репозиторіїв або сервісів. Таким чином, виникають Code Smells – порушення правил об’єктно-орієнтованого дизайну, що вперше були детально описані в книзі Мартіна Фаулера «Рефакторинг. Поліпшення існуючого коду». Нижче подаються типові порушення, що виникають внаслідок реалізації валідаційної логіки всередині типів, щовиконують операції над сутностями.

Таблиця 3.10 – Подолання Code Smells

Code Smell	Опис або підвид	Шлях подолання за допомогою бібліотеки
Блоттери	Методи та класи, які збільшились до таких пропорцій, з якими важко працювати. Великий за об’ємом код валідації.	Винесення валідаційної логіки в окремі класи-валідатори за допомогою фабрики.
Порушення принципів ООП	Switch-вирази або ж надмірні if-else конструкції. Цикломатично складна валідаційна логіка всередині класу.	Задання валідаційних предикатів через Fluent API.
Запобіжники змін	Зміни в одному місці у коді призводять до багатьох змін у інших місцях. Окремі, проте взаємопов’язані інваріанти всередині класів.	Винесення валідаційної логіки в окремі класи-валідатори за допомогою фабрики.
Надлишковий код	В першу чергу, дублювання логіки валідації у різних типах.	Винесення валідаційної логіки в окремі класи-валідатори за допомогою фабрики.
Надмірна зв’язність	Ступінь взаємозалежності між програмними артефактами.	Використання абстракцій валідаційних правил

3.7.8 Зменшення Design Smells

Робертом Мартіном було описано чотири основні симптоми «Гнилого дизайну» [11], які також відомі під назвою «Design Smells»: жорсткість, крихкість, нерухомість і в'язкість – ключові глобальні показники якості дизайну програмної системи.

Жорсткість – це тенденція ускладнення зміни програмного забезпечення, навіть у прості способи. Кожна зміна викликає каскад наступних змін у залежних модулях. Тісно пов'язана з жорсткістю крихкість. Крихкість – це тенденція ПЗ ламатися в багатьох місцях кожного разу при внесенні змін. Часто поломка відбувається у частнах, які не мають концептуальних зв'язків кодом, що змінюється. Нерухомість коду – це неможливість повторного використання програмного забезпечення з інших проектів або з частин того ж проекту. В'язкість набуває двох видів: в'язкість дизайну та в'язкість середовища. Часто існує більше одного способу зробити зміни. Деякі способи зберігають дизайн, інші ні (тобто це хаки.) Якщо методи збереження дизайну важче використовувати, ніж хаки, тоді в'язкість дизайну висока.

Усі вищеописані симптоми вирішуються за допомогою винесення логіки валідації в окремі типи, які сумісно використовуються різними частинами клієнтського коду, а також шляхом прив’язки клієнтського коду до абстракцій, а не до конкретних типів.

3.7.9 Гарантування безпеки типів

На відміну від аналогів – анотацій даних та FluentValidation, у даній бібліотеці робота з контекстом задається через делегати, що закриті типом контексту. Таким чином гарантується безпека типів на етапі компіляції, що спрощує написання коду, запобігає значній кількості помилок під час виконання та, як наслідок, пришвидшує розробку ПЗ.

3.7.10 Спрощення написання модульних тестів

По даній характеристиці бібліотека контекстуальної валідації виграє у стандартного засобу System.ComponentModel.DataAnnotations, оскільки правила перевірки задаються не у вигляді метаданих, а у формі звичаного коду. Це виключає необхідність написання модульних тестів через механізм рефлексії.

3.8 Недоліки та подальший розвиток продукту

За браком часу, значну частину описаних у Розділі 2 вимог до бібліотеки не вдалося реалізувати. Також, декілька невирішених проблем виникли у ході розробки продукту.

3.8.1 Асинхронна валідація

Наразі основним недоліком є виключно синхронна валідація. Асинхронна валідація є необхідною для роботи із контекстами, що представлені зовнішніми процесами: файлова система, база даних, інші за стосунки, кеш тощо. У теперішньому вигляді робота із такими контекстами реалізована у блокуючій манері та завдає шкоди продуктивності застосунку. У майбутній версії програми планується додати можливість задання валідаційних предикатів за допомогою асинхронних делегатів типу Func<ClassUnderValidation, DataSource, Task<bool> та асинхронних методів ValidateAsync, що повертають Task для правил, що викидають виключення, та Task<ValidationResult> для правил, що повертають результат перевірки.

3.8.2 Робота з декількома контекстами

У теперішній версії бібліотека працює виключно з одним валідаційним контекстом, хоча контекстом для однієї операції валідації може виступати декілька об’єктів. Наприклад, два або три різних Repository треба перевірити, перед тим, як зберегти сутність. Звісно, у такому випадку валідаційне правило можна зав’язати на Unit of Work. Але як бути у випадку IRepository і IService, що

водночас виступають контекстом? Можна зав’язати правило на ValueTuple<IRepository, IService>, проте зручніше працювати з валідаційними предикатами виду Func<TEntity, TContext1, TContext2, … bool>. Тому подальша версія бібліотеки повинна містити перевантаження для задання правил за допомогою усіх 16 видів Func делегатів.

3.8.3 Проблема контексту контексту

Іншим важливим недоліком є наступний. Деяку сутність TEntity треба провалідувати по контексту TContext у класах Type1 і Type2, при чому зміст валідаційних правил для кожного із цих класів повинен бути різним. Таким чином постає проблема контексту контексту. Головна проблема полягає у визначенні валідаційних правил, які треба впровадити у Type1 чи Type2 – в обох випадках вони належать до IValidatonRule<TEntity, TContext>. Рішенням є створення для Type1 і Type2 свого класу-обгортки над валідаційним правилом та впровадження цього типу, замість IValidatonRule<TEntity, TContext> у їх конструктори.

3.8.4 Шаблон Specification

Специфікація – це шаблон, який дозволяє інкапсулювати частину знань домену в єдину специфікацію та повторно використовувати її в різних частинах кодової бази. Він широко використовується у DDD, в першу чергу, для організації інваріант. Є 3 основні випадки використання для шаблону специфікації:

- пошук даних, що задовольняють специфікації, в базі;
- валідація об'єктів у пам'яті на відповідність специфікації;
- створення нового екземпляру, що відповідає критеріям.

Перші два випадки якраз підходять для контекстуальної валідації. Так, другий приклад є очевидно відповідним, а другий підходить тому, що контекст валідації, як і запит до бази, споживає перелік предикатів.

На рисунку 3.1 представлено діаграму класів шаблону специфікації.

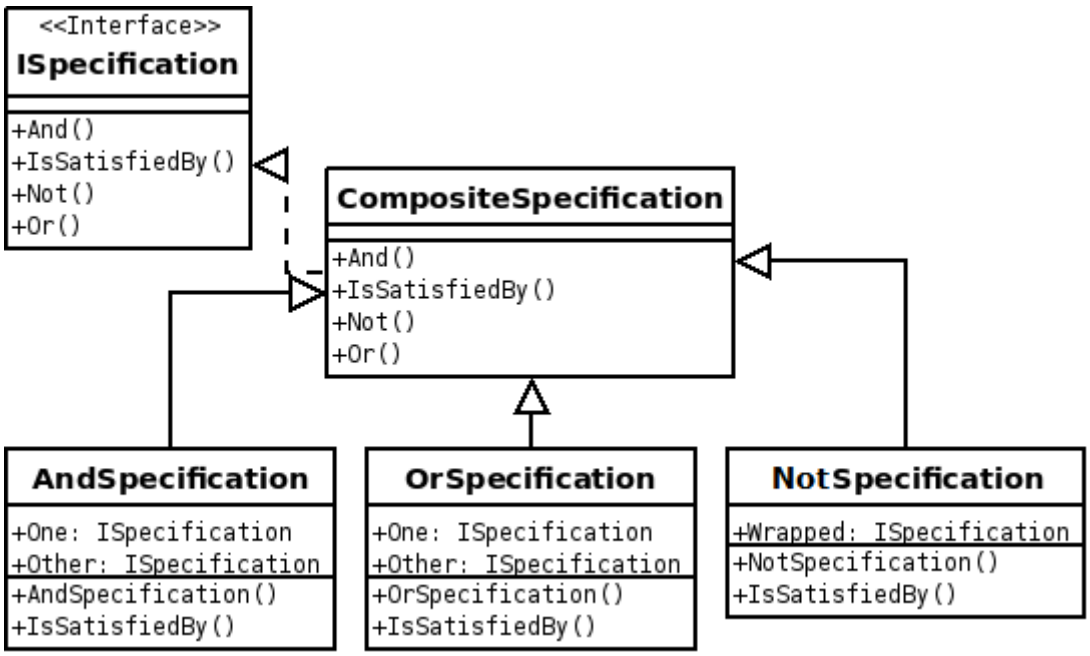

Рис. 3.1 – Шаблон Specification

У бібліотеці контекстуальної валідації необхідно реалізувати даний патерн для можливості комбінування предикатів за допомогою булевих операторів І, АБО та НЕ і, таким чином, перевикористовувати їх у різних правилах перевірки і, перш за все, описувати це у рамках Fluent API.

3.8.5 Робота із правилами FluentValidation

Наразі бібліотека підтримує роботу лише з одним зовнішнім видом валідації – атрибутами із простору імен System. ComponentModel. DataAnnotations. Проте чималою популярністю користується FluentValidation. У плани на подальший розвиток продукту входить розширити перелік зовнішніх валідаторів, додавши роботу з AbstractValidator, ValidatorAttribute, PropertyValidator, IClientValidatable та іншими видами валідаторів із даної бібліотеки, а також із типом ValidationResult.

3.8.6 Інтеграція із ASP.NET та ASP.NET Core

Інтеграція із ASP.NET та ASP.NET Core – є потужним інструментом, який використовується в аналогах: як в System.ComponentModel. DataAnnotations, так і FluentValidation. Більше того, виконання валідації моделі представлення за

допомого ModelState.IsValid у класах контролерів є офіційною рекомендацією Microsoft. Слідуючи їй, а також для того, щоб не відставати від аналогів слід розробити механізм інтеграції бібліотеки контекстуальної валідації у веб-фреймворк.

3.8.7 Можливий негативний вплив на продуктивність

У деяких випадках перевірка інваріантів у пам’яті викликає негативний вплив на продуктивність застосунку. Найчастіше контекстом сутності виступають інші сутності, які зберігаються у репозиторії. Типово, він працює із реляційною СКБД. Проблема полягає у тому, що Entity Framework працює не з делегатами, а з графом об’єктів Expression – деревами виразів, які перетворюються на SQL запит, що виконується на стороні БД сервера. Робота ж з делегатами виключає можливість перевірки інваріант на стороні БД. Можливе даної проблеми може заключатися у виражені валідаційних предикатів саме через Expression<Func>, що при цьому не інтерфейс бібліотеки.

3.9 Висновок до розділу

У цьому розділі була описана бібліотека контекстуальної валідації, що являє собою предмет науково-практичної новизни у даній роботі. Розроблена бібліотека контекстуальної валідації являє Production Rule System, що організує логіку перевірок за допомогою набору правил, кожне з яких має умову (предикат Func<TEntity, TContext, bool>) та делегат реакції на її порушення, що задається функціями NotifiedAs та ThrowingException.

Отримана бібліотека може працювати у двох режимах:

- статична валідація;
- динамічна валідація.

У ході розробки було використано різноманітні техніки:

- патерни DSL;
- прийом EIMI;

– патерни GoF.

Бібліотека має низку переваг перед аналогами:

– призводить до зменшення об’єму коду валідації;

– може бути використана у підходах відкладеної валідації та завжди валідної сутності;

– має інтуїтивно зрозумілий API;

– враховує правила System.ComponentModel.DataAnnotations;

– реалізує Exception та Notification підходи;

– сприяє зменшенню цикломатичної складності;

– сприяє подоланню Code Smells;

– сприяє зменшенню Design Smells;

– гарантує безпеки типів;

– спрощує написання модульних тестів.

За браком часу частину запланованого функціоналу не вдалося реалізувати, а деякі проблеми виплили у ході розробки. Тому бібліотека має наступні недоліки:

– відсутня асинхронна валідація;

– можливий негативний вплив на продуктивність (не через синхронність);

– відсутня робота з декількома контекстами;

– не вирішена проблема контексту контексту;

– предикати не оформлені у шаблон Specification;

– відсутня робота із правилами FluentValidation;

– відсутня інтеграція із ASP.NET та ASP.NET Core.

За результатами проведеної роботи було написано статтю «Бібліотека контекстуальної валідації у середовищі .NET», яка була опублікована у збірнику матеріалів II Міжнародної науково-практичної конференції «Інтеграція світових наукових процесів як основа суспільного прогресу» [19].

4 ВИКОРИСТАННЯ БІБЛІОТЕКИ У СЕРЕДОВИЩІ

Бібліотека контекстуальної валідації являє собою предмет науково-практичної новизни, проте вона має значення лише у застосуванні в реальній програмній або програмно-апаратній системі, що імплементована із застосуванням мови програмування C#.

Даний розділ описує систему-середовище, у якій застосовується бібліотека контекстуальної валідації та сценарії використання у даній системі.

4.1 Опис програмно-апаратного середовища

Для більш детального розуміння сценаріїв використання бібліотеки варто надати опис програмно-апаратного середовища, у якому вона використовується. Даний приклад є наближеним до реальних поширених на ринку outsource проектів.

4.1.1 Опис архітектури

4.1.1.1 Загальна архітектура системи

Середовище являє собою програмно-апаратну систему, що організована за принципами SOA.

Сервіс-орієнтована архітектура (SOA, Service Oriented Architecture) – модульний підхід до розробки програмного забезпечення, заснований на використанні розподілених, слабо пов'язаних зв'язків замінних компонентів, зі стандартизованими інтерфейсами для взаємодії за стандартизованими протоколами.

Дана архітектура була вибрана через необхідність поєднання двух платформ в одній: .NET Framework та .NET Core, а також через те, що фронт-енд застосунок являє собою SPA (Single Page Application)

Програмні комплекси, розроблені відповідно до сервіс-орієнтованої архітектури, як правило, реалізуються як набір веб-служб, які взаємодіють за

протоколом SOAP, але існують і інші реалізації (наприклад, на базі jini, CORBA, на основі REST).

Інтерфейси компонентів у сервісно-орієнтованої архітектурі інкапсулюють деталі реалізації (операційну систему, платформу, мову програмування) від інших компонентів, тим самим забезпечуючи комбінування та багаторазове використання компонент для побудови складних розподілених програмних комплексів, забезпечуючи незалежність від використовуваних платформ та інструментів розробки, що сприяють масштабуванню та підтримуваності створюваних систем.

Як видно на доданій до роботи структурній діаграмі, система складається із нижче описаних частин, які являють собою окремі процеси, по суті, окремі застосунки, проте, з точки зору користувача, ведуть себе як єдиний застосунок. Перелік підсистем:

- підсистема моніторингу;
- сервіс моніторингу;
- база даних основного сервісу;
- основний сервіс;
- Redis Cache;
- сервіс репортингу;
- Angular клієнт;
- черга повідомлень Rabbit MQ;
- сервіс логування;
- сервіс авторизації.

4.1.1.2 Архітектура модулів

У кожній підсистемі модулі програми організовані за стилем Onion Architecture. Стиль Onion Architecture призводить до більш підтримуваних програм, оскільки він підкреслює розмежування проблем у всій системі

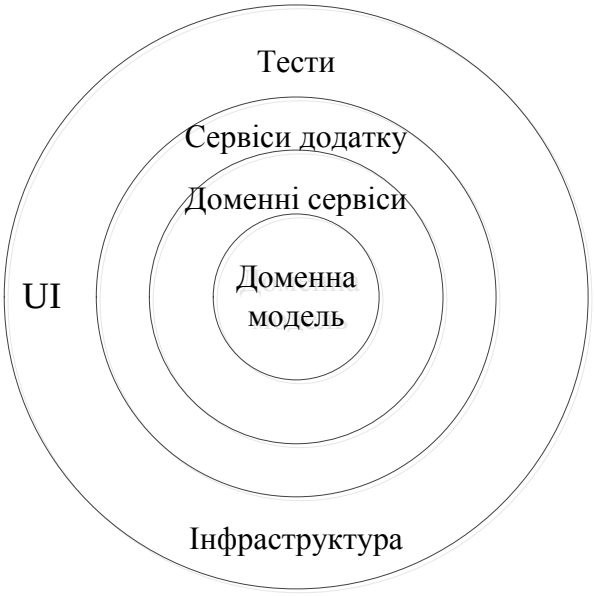

Рис. 4.1 – Архітектура цибулини

Діаграма зверху зображує Onion Architecture. Основний посил полягає в тому, що вона контролює зв'язність. Основне правило полягає в тому, що весь код може залежати від шарів більш центральних, але від шарів далі від ядра. Інакше кажучи, вся зв'язність напрямлена до центру.

У самому центрі – Модель домену, яка представляє собою комбінації стану та поведінки, яка моделює істину. Навколо моделі домену є інші шари з більшою поведінкою. Кількість шарів в ядрі програми буде різною, однак головне, що доменна модель є центром, і оскільки вся зв'язність знаходиться в центрі, доменна модель пов'язана лише з собою. Перший шар навколо моделі домену, як правило, містить інтерфейси, що забезпечують збереження та отримання об'єктів, що називаються інтерфейсами репозиторіїв. Однак поведінка збереження об'єкта не в ядрі програми, тому що вона включає взаємодію із базою даних. Лише інтерфейс знаходиться в ядрі програми. На краях – користувальницький інтерфейс, інфраструктура та тести. Зовнішній шар призначений для речей, які часто змінюються. Ці речі слід навмисно ізолювати від ядра. На краю – клас, який реалізує інтерфейс сховища. Цей клас пов'язаний із певним методом доступу до даних, і саме тому він знаходиться поза ядром програми. Цей клас реалізує інтерфейс сховища і тим самим з ним пов'язаний.

Onion Architecture багато в чому залежить від принципу Dependency Inversion. Ядро додатку потребує реалізації основних інтерфейсів, і якщо ці класи-виконавці знаходяться на краях програми, потрібен певний механізм для введення цього коду під час виконання.

База даних – не центральна, вона зовнішня. Винесення бази даних може бути досить суттєвою зміною, оскільки деякі люди думають про програми як "програми для баз даних". З "Onion Architecture" немає програм для баз даних. Є програми, які можуть використовувати базу даних як службу зберігання, але лише за умови, що деякий код зовнішньої інфраструктури реалізує інтерфейс, який має сенс для ядра додатку. Розмежування програми і бази даних, файлової системи тощо, знижує витрати на обслуговування.

Ключова відмінність полягає в тому, що будь-який зовнішній шар може безпосередньо викликати будь-який внутрішній шар. З традиційно багатошаровою архітектурою шар може лише викликати шар безпосередньо під ним. Це один з ключових моментів, який робить "Onion Architecture" відмінною від традиційної багатошарової архітектури. Інфраструктура витісняється до країв, що не зв’язує код бізнес-логіки. Код, який взаємодіє з базою даних, реалізовує інтерфейси із ядра програми. Ядро програми зв'язано з цими інтерфейсами, але не з фактичним кодом доступу до даних. Таким чином, ми можемо змінювати код у будь-якому зовнішньому шарі, не впливаючи на ядро програми. Ми включаємо тести, тому що будь-який довго-живий додаток потребує тестів. Тести на краях цибулини, оскільки ядро програми не поєднується з ними, але тести пов'язані з ядром програми. Ми також можемо мати ще один шар тестів навколо всього зовнішнього середовища, коли ми перевіримо інфраструктурну обв’язку та інтерфейс.

Цей підхід до архітектури додатків гарантує, що ядро додатків не повинно змінюватися, при: зміні інтерфейсу користувача, зміні доступу до даних, зміні веб-сервісів та інфраструктури повідомлень, зміна способів введення-виведення.

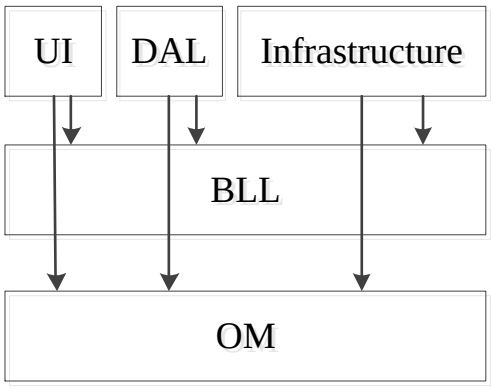

Рис. 4.2 – Архітектура цибулини пласка

Згори, я створив діаграму, яка намагається показати, як Onion Architecture буде виглядати, коли вона представлена як традиційна шарувата архітектура. Велика різниця в тому, що Data Access - це верхній шар разом з інтерфейсом користувача, введенням / виводом тощо. Ще одна важлива відмінність полягає в тому, що наведені вище шари можуть використовувати будь-який шар під ними, а не лише шар, що знаходиться безпосередньо під ним. Крім того, бізнес-логіка пов'язана з об'єктною моделлю, а не з інфраструктурою.

Основні принципи Onion Architecture:

– програма побудована навколо незалежної об'єктної моделі;

– внутрішні шари визначають інтерфейси. Зовнішні шари реалізують інтерфейси;

– напрямок зв’язності до центру;

– весь основний код програми може бути скомпільований та запущений окремо від інфраструктури.

4.1.1.3. Організація Dependency Injection

Dependency Injection – це шаблон проектування, в якому сервіси реалізовують інтерфейси, які в єдиному місці привязуються до конкретної їх реалізації і потім, при звернені до відповідного інтерфейсу, підставляється його реалізація.

Шаблон дозволяє досягти слабкой звязаності і притримуватись принципів інверсіїї залежностей та єдиного обвязку.

Інверіся залежностей – абстрактний принцип, який полягає в ізоляції компонентів друг від друга, не привязуючись до конкретних реалізацій.

Принцип єдиного обовязку полягає в тому, що кожен компонент повинен виконувати лише свої функції та задачі, які він повинен повністю інкапсулювати.

Переваги даного шаблону проектування:

– обєкти системи стають більш незалежними, в результаті чого упрощується написання юніт-тестів. Це відбувається за рахунок використання макетів обєктів, які імітують інші обєкти.

– впровадження залежностей сприяють паралельній розробці. Над розробкою модулів, можна працювати незалежно, тому що не потрібна реалізація інших модулів програми, а достатньо лише їх опис за допомогою інтерфейсів.

– перенесення деталей конфігурації системи в конфігураційні файли.

– підміна реалізацій. Можлива заміна реалізацій сервісів, без зміни залежностей.

– IoC-контейнер – це якась бібліотека чи фреймворк, яка дозволяє реалізувати патерн Dependency Injection. В ASP.NET Core додатках використовується нативний контейнер, у додатках ASP.NET – Ninject.

4.1.2 Back-end платформа

У якості основної платформи для розробки серверної частини застосунку було обрано ASP.NET Core – крос-платформний, високопродуктивний фреймворк з відкритим кодом для створення сучасних, хмарних, Інтернет-підключених додатків. ASP.NET Core – це редизайн ASP.NET 4.x, із архітектурними змінами, що призводять до більш гнучкої, модульної структури.

ASP.NET Core надає наступні переваги:

– єдине джерело створення веб-інтерфейсів.

– архітектура призначена для тестування.

– Razor Pages робить сценарії, орієнтовані на кодування, більш простими та продуктивними.

– можливість розробки та запуску в Windows, MacOS та Linux.

– Open-source та орієнтованість на розробників.

– інтеграція сучасних клієнтських фреймворків та процесів розробки.

– орієнтована на хмарні додатки система конфігурації.

– вбудований dependency injection.

– легкий, високопродуктивний та модульний конвеєр обробки HTTP-запитів.

– можливість хостингу в IIS, Nginx, Apache, Docker або self-host у власному процесі.

– інструментарій, що спрощує сучасний веб-розробку.

Так, відповідно до офіційних рекомендацій Microsoft, .NET Core варто використовувати для серверної частини програми, коли:

– існують крос-платформні потреби;

– додаток має мікросервісну або іншу сервіс-орієнтовану архітектуру;

– використовуються контейнери Docker;

– потрібні високопродуктивні та масштабовані системи.

Окрім платформи ASP.NET Core у застосунку також використовується .NET Framework для вашої серверної програми, оскільки використовуються сторонні бібліотеки .NET або NuGet, недоступні для .NET Core (EPPlus) та технології .NET, недоступні для .NET Core.

4.1.3 Підсистема моніторингу

Розробка підсистеми моніторингу виходить за рамки даної роботи. Проте для повноти картини варто додати її короткий опис. Дана підсистема складається із наступних елементів:

– датчиків, що фіксують зміни показників будівлі: вологість, температура, рух шкідників, рівень води тощо;

– контролерів, котрі об’єднують групи датчиків за територіальною ознакою;

– допоміжної мережевої апаратури (термінатори, репітери, трансивери тощо).

Кожен контролер, отримавши показники датчиків, які відрізняються від попередніх, надсилає повідомлення із показниками у мережу на IP-адресу та порт, який слухає сервіс моніторингу.

4.1.4 Сервіс моніторингу

Даний сервіс являє собою self-hosted (тобто таку, що не потребує сервера, а розгортається у власному процесі) WCF службу. Задача сервісу моніторингу полягає у наступному:

– прослуховування запитів, що йдуть від підсистеми моніторингу;

– попередній аналіз отриманих даних (віднесення їх до хибних – таких, що вказують на несправність підсистеми моніторингу, дійсних та несуттєвих, тобто таких, що знаходяться межах норми і не потрибують запису в БД);

– пінгування підсистеми моніторингу (health-check) та повідомлення основного сервісу в разі несправності;

– передача відповідних аналітичних або отриманих на основний сервіс.

4.1.5 Основний сервіс

Даний сервіс являє собою ASP.NET Core 2.2 Web API застосунок, що є центральним елементом системи та виконує роль сервісу-фасаду для Angular 2+ клієнту. До його функцій входять:

– створення, видалення, редагування працівників;

– облік датчиків та контролерів, їх стану, розміщення тощо;

– накопичення та аналіз даних про параметри будівлі;

– внесення та зберігання даних про будівлю (креслення, комунікації, стан елементів будівлі (крівля, підвал, стіни тощо));

– формування задач для працівників на основі їх зайнятості, кваліфікації та характеристик стану будівлі;

– контроль за життєздатністю підсистеми моніторингу та своєчасне повідомлення персоналу про потенційні поламки;

– зв’язка Angular 2+ клієнту із іншими сервісами.

4.1.6 База даних основного сервісу

У якості СКБД обрано MS SQL Server 2012 – реляційну СКБД, яка офіційно підтримується Microsoft та має найкращу підтримку роботи із ORM Entity Framework, через яку організована взаємодія із БД. Бізнес логіка взаємодіє безпосередньо із абстракціями патерну Repository, у реалізаціях яких інкапсульована логіка роботи безпосередньо з ORM.

Повна схема даних основного сервісу представлена у переліку креслень на ER-діаграмі. Нижче наведено опис таблиць схеми.

Таблиця 4.1 – Clients (Клієнти)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
ClientFullName	nvarchar	Повне ім’я клієнта	UQ
CompanyName	nvarchar	Назва компанії	
Phones	nvarchar	Список телефонних номерів	

Таблиця 4.2 – Contracts (Контракти – договори про надання послуг)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
ContractNumber	nvarchar	Номер договору про надання послуг	UQ
ClientId	nvarchar	Id клієнта	FK (Clients)
MonthlyFee	decimal	Розмір щомісячної оплати послуг	
Balance	decimal	Баланс клієнта	
SignedDate	date	Дата підписання договору	

Таблиця 4.3 – Buildings (Будівлі)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
City	nvarchar	Місто	
Street	nvarchar	Вулиця	
BuildingNumber	int	Номер будинку	
ContractId	int	Id контракту	FK (Contracts)

Таблиця 4.4 – Placements (Розміщення)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
BuildingId	nvarchar	Id будівлі	FK (Buildings)
PlacementTypeId	nvarchar	Id типу розмішення	FK (PlacementTypes)
FasteningTypeId	int	Id типу кріплення	FK (FasteningTypes)
Entrance	int	Номер під’їзду	

Таблиця 4.5 – BuildingPlanPlacements (Розміщення на плані будівель)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
BuildingPlanId	int	Id плану будівлі	FK (BuildingPlans)
PlacementId	int	Id розміщення	FK (PLacements)
XCoord	int	Координата X	NN
YCoord	int	Координата Y	NN

Таблиця 4.6 – MeasurementTypes (Типи замірів)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
Type	nvarchar	Тип виміру	NN, UQ
SensorTypeId	Int	Id типу датчика	FK (SensorTypes)

Таблиця 4.7 – FasteningTypes (Типи кріплень)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
Type	nvarchar	Тип кріплення	NN

Таблиця 4.8 – PlacementTypes (Типи розміщень)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
Type	nvarchar	Тип розміщення	NN

Таблиця 4.9 – ParameterNorms (Норми параметрів)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
PlacementId	int	Id розміщення	FK (Placements)
PlacementTypeId	int	Id типу розміщення	FK (PlacementTypes)
MeasurementTypeId	int	Id типу виміру	FK (MeasurementTypes)
From	decimal	Нижня межа	NN
To	decimal	Верхня межа	NN

Таблиця 4.10 – SensorTypes (Типи датчиків)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
Type	nvarchar	Тип датчика	NN, UQ

Таблиця 4.11 – DetectionTypes (Типи детекцій)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
Type	nvarchar	Тип	NN
SensorTypeId	int	Тип датчика, що може зафіксувати даний тип	FK (SensorTypes)

Таблиця 4.12 – Controllers (Контролери)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
IpAddress	int	IP адреса	NN, UQ
PlacementId	int	Id розміщення	FK (Placements)
State	int	Стан контролера (працює, зламаний, невстановлено)	

Таблиця 4.13 – Sensors (Датчики)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
SensorTypeId	int	Id типу датчика	FK (SensorTypes)
PlacementId	int	Id розміщення	FK (Placements)
ControllerId	int	Id контролера, за яким закріплено датчик	FK (Controllers)
State	int	Стан датчика (працює, зламаний, невстановлено)	NN

Таблиця 4.14 – SpecializationsDetectionTypes (Розподіл типів детекцій по спеціальностях)

Назва колонки	Тип даних	Зміст	Обмеження
SpecializtionId	int	Id спеціальності	PK, FK (Specializtion)
DetectionTypeId	int	Id типу детекції	PK, FK (DetectionType)

Таблиця 4.15 – SpecializationsMeasurementTypes (Розподіл типів заміру по спеціальностях)

Назва колонки	Тип даних	Зміст	Обмеження
SpecializtionId	int	Id спеціальності	PK, FK (Specializtion)
MeasurementTypeId	nvarchar	Id типу заміру	PK, FK (MeasurementType)

Таблиця 4.16 – Workers (Працівники)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
ContractNumber	nvarchar	Номер трудового договору	UQ
LastName	nvarchar	Прізвище	NN
FirstName	nvarchar	Ім’я	NN
MiddleName	nvarchar	По батькові	NN
SpecializationId	int	Id спеціальності	FK (Specializations)
QualificationLevel	nvarchar	Кваліфікаційний рівень	NN
Salary	decimal	Заробітна плата	NN
CurrentBonus	decimal	Теперішній розмір премії	

Таблиця 4.17 – SystemTasksWorkers (Розподіл задач по працівникам)

Назва колонки	Тип даних	Зміст	Обмеження
WorkerId	int	Id працівника	PK, FK (Workers)
SystemTaskId	int	Id задачі	PK, FK (SystemTasks)

Таблиця 4.18 – SystemTasks (Задачі)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
Ocurred	date	Час виникнення	NN
Solved	date	Час вирішення	
Description	nvarchar	Опис	NN

Таблиця 4.19 – SystemTaskMeasurements (Відношення задач до вимірів)

Назва колонки	Тип даних	Зміст	Обмеження
MeasurementId	int	Id заміру	PK, FK (Measurements)
SystemTaskId	int	Id задачі	PK, FK (SystemTasks)

Таблиця 4.20 – Detections (Детекції)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
SensorId	int	Id датчика	FK (Sensors)
DateTime	date	Час та дата виникнення	NN

Таблиця 4.21 – ElevatorParameterMeasurements (Заміри параметрів ліфта)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
SensorId	int	Id датчика	FK (Sensors)
Velocity	decimal	Швидкість (м/с)	NN
Acceleration	decimal	Прискорення ($м/с^2$)	NN
Weight	decimal	Вага (кг)	NN
DateTime	date	Час та дата заміру	NN

Таблиця 4.22 – BuildingPlans (Плани будівель)

Назва колонки	Тип даних	Зміст	Обмеження
Id	int	Id	PK
Name	nvarchar	Назва	
ImagePath	nvarchar	Розміщення у файловій системі	

4.1.7 Redis Cache

Redis – це сховище даних в оперативній пам’яті, яке у даному застосунку використовується кеш-пам'ять та розгортається як окремий процес. Він підтримує структури даних, такі як рядки, хеші, списки, набори, сортовані множини з діапазоном запитів тощо.

Даний кеш використовується для передачі даних від основного сервісу на сервіс репортингу та передачі від останнього до основного масиву байт, що являють собою Excel-репорт. Робота з Redis Cache в основному сервісі

організована через вбудовані в ASP.NET Core модулі, а в сервісі репортингу за допомогою бібліотеки ServiceStack.Redis.Complete.

4.1.8 Сервіс репортингу

Сервіс являє собою ASP.NET Web API застосунок, який хоститься на сервері IIS 7. Із сервісом авторизації він взаємодіє за допомогою OWIN middleware. З базою даних не взаємодіє. Дані для генерації репортів споживає з Redis Cache. Ключовим модулем є власне модуль генерації репортів описаний нижче.

Генерація звітів у форматах .xlsx (MS Excel) та .docx (MS Word) у серверних додатках є досить розповсюдженою задачею. Існує декілька стандартних способів реалізації даного завдання.

4.1.8.1 Взаємодія із запущеним процесом Excel застосунку

Використання COM через Microsoft.Interop.Services – пряма взаємодія із запущеним процесом застосунку MS Excel чи MS Word через код, що не керується (unmanaged code). Microsoft.Interop.Services надає перелік типів, що є обгортками над COM-компонентами – некерованими класами C++, котрі через Win32 API стартують необхідний процес офісного додатку та передають команди на, наприклад, створення нової колонки, параграфа, заповнення комірки таблиці тощо. Отже, генерація .xlsx / .docx файлу відбувається безпосередньо у офісному додатку. Це передбачає встановлення Microsoft Office на сервері. До того ж даний підхід є доволі громіздким, має проблеми із безпекою потоків, погано працює на платформах x64 і не працює взагалі на Linux-серверах.

4.1.8.2 LINQ to XML

Генерувати файли «вручну». .xlsx та .docx файли зберігаються у форматі OfficeOpenXML – це ніщо інше як xml файли розкидані по спеціальним папкам (стилі, шаблони, текстовий зміст, графіки, формули тощо) запаковані у zip-архів. Таким чином за допомогою System.IO та LINQ to XML можна реалізувати генерацію необхідних документів без залежності на бібліотеки третіх сторін та роботи з COM. Також можна бути впевненим у тому, що у середовищі ASP.NET

Core, а отже і на серверах окрім Windows даний підхід працюватиме. Проте таким чином доведеться писати доволі багато коду, розбиратися окремо із кожним форматом файлів, особисто відслідковувати усі зміни форматів та вносити відповідні зміни у код.

4.1.8.3 OpenXML SDK

Генерація файлів за допомогою Microsoft OpenXML SDK. Даний підхід є офіційною рекомендацією Microsoft. Утиліта Open XML SDK 2.0 Productivity Tool дає можливість перетворити готовий файл у код, що його згенерує, порівняти декілька .docx / .xlsx файлів у термінах строк xml або C# коду. Автоматична генерація необхідного коду не вирішую проблему до кінця, оскільки файл треба наповнити динамічними даними (наприклад, із БД). Отже, розробнику все одно доведеться напряму працювати із OpenXML API. Код у такому випадку також виходить громіздким та важкопідтримуваним.

4.1.8.4 OpenXML PowerTools

Використання фреймворку OpenXML PowerTools, що є надбудовою над Microsoft OpenXML надає більш зручний API, проте все одно занадто складний для простої генерації репортів.

4.1.8.5 Використання спеціалізованих бібліотек

Таблиця 4.23 – Рейтинг бібліотек, що працюють із Excel файлами, за популярністю [20]

Назва	Платформа	Кількість завантажень	Дата останнього оновлення
EPPlus	.NET Framework .NET Standard	5,21 млн	30.05.2018
Excel Data Reader	.NET Framework .NET Standard	2,09 млн	15.10.2018
Closed XML	.NET Framework .NET Standard	1,91 млн	07.08.2018

Продовження таблиці 4.23.

Назва	Платформа	Кількість завантажень	Дата останнього оновлення
NPOI	.NET Framework .NET Standard	1,88 млн	28.10.2018
LINQ to Excel	.NET Framework	424 тис	19.02.2017

Модуль excel-репортів було виконано із використанням бібліотеки EPPlus через популярність та попередній досвід роботи із нею. Діаграма класів даного модулю додана до переліку креслень.

4.1.9 Angular 2+ клієнт

Хоча Angular клієнт і не впливає на використання бібліотеки контекстуальної валідації, для повноти картини варто його коротко описати.

Клієнт являє собою SPA, що хоститься на liveserver. Додаток складається із набору lazy-load модулів, які розбиті, спершу, по ролях (адміністратор, користувач, працівник), а потім по функціоналу. Автентифікація та авторизація на стороні клієнтського додатку організована за допомогою Guard-класів, що імплементують інтерфейси CanActivate та CanActivateChild, взаємодія з сервісом авторизації IdentityServer 4 відбується за допомогою бібліотеки open-id-connect-client та JWT. Для організації верстки використовувався готовий Bootstrap шаблон, набір готових шрифтів та іконок fontawesome та бібліотека ng-bootstrap.

4.1.10 Сервіс логування

У середовищі .NET можна легко логувати події у Windows Event Viewer без використання додаткових бібліотек. Це максимальна зручність та надійність. Проте написання в Event Viewer не завжди працює для деяких програм, наприклад, ви можете написати записи про події у файл журналу, щоб поділитися ним з іншими відділами або аналізування журналів за допомогою спеціальних

засобів таких як SolarWinds Loggly, Elastic Search, Splunk. Більше того у середовищі .NET Core Windows Event Viewer може бути недоступним взагалі.

Кілька вендорів фреймворків логування обіцяють просте налаштування та високу швидкість роботи. Усі нижчерозглянуті фреймворки мають приблизно однаковий за зручністю API, проте відрізняються за набором функцій та швидкодією. На рисунках 4.3., 4.4. та 4.5. представлено Benchmark тест найбільш популярних фреймворків у середовищі [21], яке є репрезентативним у реальному світі – сервер, де розміщуються три веб-застосунки, SQL Server і MySQL для трьох програм. Щоб імітувати завантаження ЦП, було встановлено окремий поточний процес, який генерував прості числа при тестуванні та запуску. Виробничий сервер – Windows Server 2008 з оперативною пам'яттю 8 Гб та процесором Intel i5 з чотирма ядрами, одне ядро на кожен потік. Програми були створені за допомогою Visual Studio 2012. Log4net, NLog, Microsoft Enterprise Library та NSpring були встановлені в консольних програмах. ELMAH є веб-логером, тому ця програма була запущена з веб-додатком у .NET 4.5.

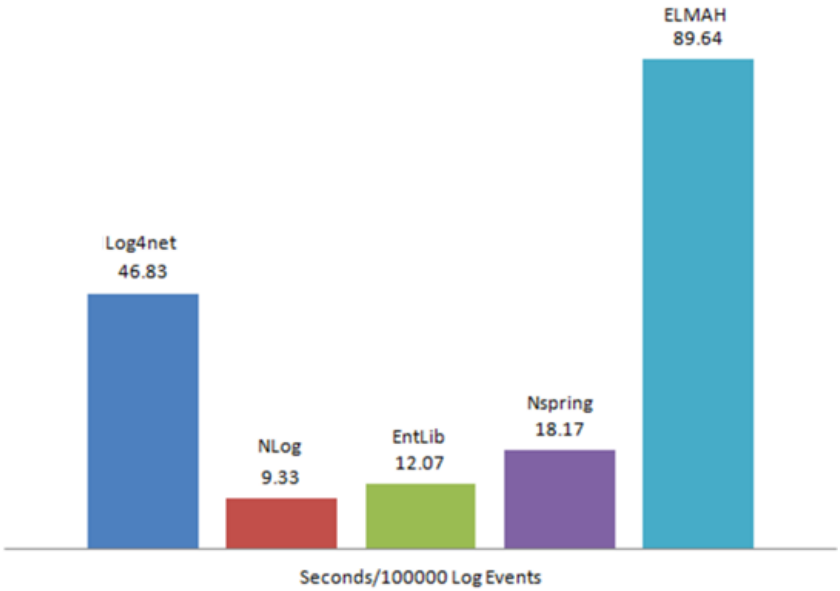

Рис. 4.3 – Benchmark тест фреймворків легування

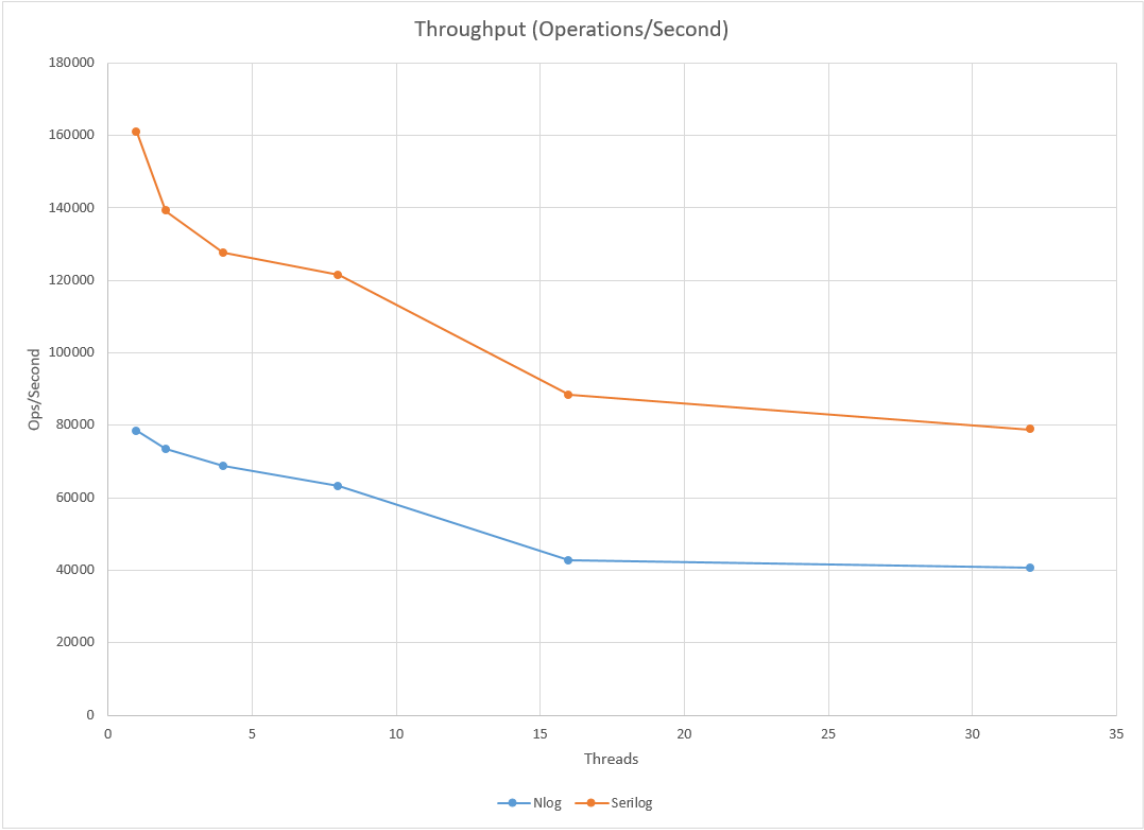

Рис. 4.4 – Операції запису у файл для кількох потоків

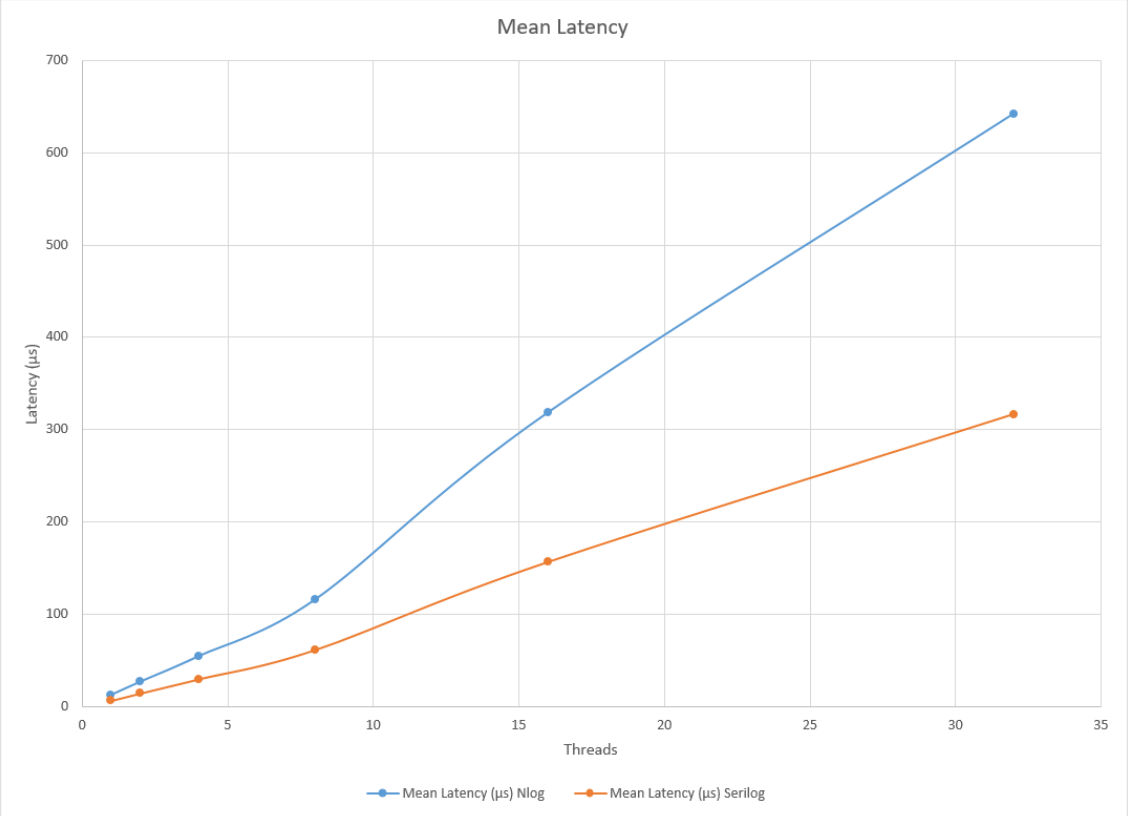

Рис. 4.5 – Середня затримка перед операцією

З результатів ясно, що хоча Serilog і NLog аналогічно налаштовані, результати для Serilog набагато кращі, ніж ті, що використовуються для NLog у пропускній здатності та затримці. У більшості випадків затримка для Serilog у два рази менша, ніж затримка для NLog, а пропускна спроможність в два рази більше. Решта фреймворків легування ще більше відстають від Serilog у термінах швидкодії.

Крім того, Serilog має якісні переваги у плані набору функції:

– сучасний та зручний підхід до конфігурації – Fluent API;

– велика кількість Sink-ів (плагінів / цілей), які можна використовувати паралельно (напр., [22] Dynamo DB, MS SQL, NLog, Rabbit MQ, Splunk, File, Console);

– шаблони повідомлень;

– структурне логування (об'єкт JSON із усіма властивостями, що були передані логеру);

– концепція збагачення, за допомогою якої збагачувачі використовуються для додавання додаткової інформації до записаних даних.

4.1.11 Сервіс авторизації

Для даної архітектури авторизацію краще вього організувати за допомогою SSO із окремим сервісом авторизації. SSO надає значні переваги с точки зору користувачів. Використання пов'язаних ідентифікацій означає, що їм потрібно керувати лише одним логіном і паролем для пов’язаних веб-сайтів. Користувальницький досвід для них стає кращим, оскільки вони можуть уникнути декількох логувань в по суті єдиній системі. Облікові дані користувача (одиничний набір) будуть зберігатися в одній базі даних, а не декілька облікових даних у декількох базах даних (разом із, доволі ймовірно, повторюваними паролями). Це також означає, що розробникам різних програм не потрібно зберігати паролі. Замість цього вони можуть прийняти докази ідентифікації або дозволу з надійного джерела.

Є декілька рішень для реалізації SSO. Три найбільш часто використовуваних протоколи веб-безпеки – OpenID, OAuth та SAML. Реалізації та бібліотеки вже існують на кількох мовах, а стандартизований протокол забезпечує кращу сумісність, ніж індивідуальне рішення.

OpenID – відкритий стандарт аутентифікації, який пропагується некомерційною організацією OpenID Foundation. Станом на березень 2018 року в Інтернеті існує понад мільярд облікових записів з підтримкою OpenID, а організації, такі як Google, WordPress, Yahoo і PayPal, використовують OpenId для автентифікації користувачів. Користувач повинен отримати обліковий запис OpenID через постачальника ідентифікаторів OpenID (наприклад, Google). Тоді користувач використовуватиме цей обліковий запис для входу на будь-який веб-сайт (залежна сторона), який приймає аутентифікацію OpenID. Стандарт OpenID являє собою основу для комунікації, яка повинна відбуватися між постачальником ідентифікатора та залежною стороною.

OAuth2 забезпечує безпечний делегований доступ, тобто програма, яка називається клієнтом, може приймати дії або отримувати ресурси на сервері ресурсів від імені користувача, без того, що користувач надає свої облікові дані додатку. OAuth2 реалізує це, за допомогою спеціальних маркерів (token), що видаваються постачальником ідентифікаторів (identity provider) стороннім додаткам за погодження користувачем. Потім клієнт (застосунок) використовує маркер для доступу до сервера ресурсів від імені користувача.

SAML (Security Assertion Markup Language) – найстаріший стандарт із усіх трьох, який був розроблений ще в 2001 р., Найновіше серйозне оновлення було у 2005 р. Подібно до термінології двох інших стандартів, SAML визначає принципала (principal) – кінцевого користувача, який намагається отримати доступ до ресурсу. Існує постачальник послуг, який є веб-сервером, до якого принципал намагається отримати доступ. І є постачальник ідентифікаційних даних, який є сервером, який містить ідентифікатори та облікові дані принципала.

Таблиця 4.24 – Порівняльна характеристика протоколів авторизації та автентифікації

Характеристика	OAuth2	OpenId	SAML
Формат токена	JSON або SAML2	JSON	XML
Авторизація	+	-	+
Автентифікація	Псевдо-автентифікація	+	+
Рік заснування	2005	2006	2001
Поточна версія	OAuth2	OpenID Connect	SAML 2.0
Передача даних	HTTP	HTTP GET та HTTP POST	HTTP Redirect (GET) прив’язка, SAML SOAP прив’язка, HTTP POST прив’язка та ін.
Слабкі місця в безпеці	Фішинг. OAuth 2.0 не підтримує підпис, шифрування, зв'язування каналів або перевірку клієнта. Натомість він повністю покладається на TLS за конфіденційність.	Фішинг. Провайдери ідентифікації мають журнал реєстраційних даних OpenID, що робить збитковий обліковий запис більшим порушенням конфіденційності	Огортання XML підпису надає можливість представитися будь-яким користувачем

Продовження таблиці 4.24.

Характеристика	OAuth2	OpenId	SAML
Найкраще підходить для:	Авторизація API	Єдиний вхід (SSO) для споживчих додатків	Єдиний вхід для ентерпрайз додатків. Не підходить для мобільних пристроїв

OpenID Connect – це стандарт, що додає автентифікацію до OAUTH2 – стандарту, який призначений лише для авторизації. OIDC додає автентифікацію, вводячи поняття токена ідентифікатора, який є JWT, що забезпечує підписаний доказ автентифікації користувача. Отже, OIDC поєднує в собі усі переваги вищеописаних протоколів. Саме тому за протокол авторизації було обрано OpenID Connect.

4.1.12 Тестування системи

У ході розробки системи юніт-тести були організовані відповідно до підходу «Arrange-Act-Assert», який став, практично, стандартом для всієї галузі. Він передбачає розділення тестових методів на три частини: підготовка тестових даних (Arrange), виконання дії (Act), що підлягає тестуванню, і перевірка отриманих результатів або стану об’єкту (Assert).

4.1.12.1 Тестування back-end частини

Юніт-тести back-end частини системи були написані за допомогою нижчезазначених засобів.

NUnit – відкрите середовище модульного тестування застосунків для .NET. Воно було перенесене з мови Java (бібліотека JUnit). Перші версії NUnit були написані на J#, але потім весь код був переписаний на C# з використанням таких нововведень .NET, як атрибути.

Існують також відомі розширення оригінального пакету NUnit, значна частина з них також з відкритим сирцевим кодом. NUnit.Forms доповнює NUnit засобами тестування елементів користувацького інтерфейсу Windows Forms. NUnit.ASP виконує цю саму задачу для елементів інтерфейсу в ASP.NET.

AutoFixture – це бібліотека з відкритим вихідним кодом для .NET, призначена для мінімізації фази "Arrange" у юніт-тестах, щоб максимізувати підтримуваність тестового коду. Її основна мета полягає в тому, щоб дозволити розробникам зосередитися на тестуванні, а не на тому, як налаштувати тестовий сценарій, полегшуючи створення графу об'єктів, що містять тестові дані.

AutoFixture призначена для того, щоб зробити Test-Driven Development більш продуктивним а юніт-тести більш стійкими до рефакторингу. Це реалізовано, шляхом видалення потреби в ручному кодуванні анонімних змінних як частину фази встановлення тесту. Серед інших функцій, вона пропонує загальну реалізацію шаблону Test Data Builder.

4.1.12.2 Тестування front-end частини

Юніт-тести front-end частини системи були написані за допомогою нижчезазначених засобів.

Jasmine – BDD фреймворк (Behavior-Driven Development, Розробка на основі поведінки) для тестування JavaScript коду, що запозичив багато рис з RSpec. Тести на ньому мають бути простим для читання. Простий світовий тест виглядає як наведений нижче код, де description () описує набір тестів, і () - це індивідуальна специфікація тесту. Назва "it ()" слідує ідеї розвитку, керованого поведінкою, і служить першим словом в імені тесту, яке повинно бути повним реченням. Використання слідувати синтаксису, подібному до RSpec.

Karma Test Runner є інструментом, який створює веб-сервер, який виконує вихідний код проти тестового коду для кожного з підключених браузерів. Результати кожного тесту для кожного браузера відображаються через командний рядок розробнику таким чином, що вони можуть бачити те, які браузери та тести пройшли або які не змогли пройти. Браузер також може бути захоплений вручну, відвідавши URL-адресу, де прослуховує сервер Karma, або автоматично,

дозволяючи Кармі знати, які браузери почнуть запускати, коли запускається Карма (див. браузери). Карма також дивиться всі spec-файли, вказані в файлі конфігурації, і коли будь-який файл змінюється, він запускає тестовий пробіл, відправляючи сигнал серверу тестування, щоб інформувати всі захоплені браузери, щоб знову запустити тестовий код. Кожний браузер потім завантажує вихідні файли всередину IFrame, виконує тести і звітує результати на сервер.

4.2 Опис сценаріїв використання бібліотеки

Бібліотека контекстуальної валідації використовується в описаній вище системі. Необхідність перевірки інваріант виникає в ній перед здійсненням операцій, що змінюють стан системи. Це може бути як робота в межах оперативної пам’яті окремого процесу, так і складна міжпроцесна взаємодія. У таблиці 4.25 наводиться перелік різнотипних застосувань даної бібліотеки.

Таблиця 4.25 – Опис контекстуальних валідацій у системі

Сценарій валідації	Контекст	Сутність	Потребує асинхронності
Користувач має декілька облікових записів, при цьому усі вони мають однаковий маркер юзера. Переконатися, що користувач ще не має сесії під жодним зі своїх облікових записів.	CurrentThread. Principal	Модель логіну, що надійшла з Angular 2+ клієнту	-
Перед внесенням змін до БД основного сервісу, треба переконатися, що існує відповідний запис у БД сервісу авторизації.	REST API клієнт сервісу авторизації	Сутність, що вноситься в БД основного сервісу	+

Продовження таблиці 4.25.

Сценарій валідації	Контекст	Сутність	Потребує асинхронності
Ключем у Redis Cache виступає SHA256 хеш об’єкту, у вигляді масивів байт. Треба переконатися, що об’єкт, аналогічний тому, який ми збираємося записати у кеш, не існує.	Redis Cache	Сутність, яку треба записати в кеш	+
Із сервісу моніторингу прийшов пакет із IP-адресою контролера. Перед записом даних в БД, треба переконатися, що IP-адреса валідна.	Репозиторій основного сервісу	Пакет із повідомленням від контролера	+
У пам’яті закешовані норми параметрів будівлі. Треба переконатися, що дані, котрі надійшли від сервісу моніторингу задовольняють норми.	Перелік норм параметрів	Пакет із повідомленням від контролера	-
З Angular 2+ клієнту надходять одночасно картинка, що представляє об’єкт BuildingPlan та масив об’єктів Building Plan Placement. Переконатись, що координати розміщень не виходять за межі плану	BuildingPlan	BuildingPlanPlacement	-

Продовження таблиці 4.25.

Сценарій валідації	Контекст	Сутність	Потребує асинхронності
Від клієнта надходить запит на переведення SystemTask у стан виконано. Перевірити, чи закріпленя задача ще за кимось із вищим кваліфікаційним рівнем. Якщо так – викинути виключення.	Репозиторій основного сервісу	SystemTask	+
Від клієнта надходить запит на відправленя працівника у відпустку. Переконатися, що є хтось цієї ж спеціальності не нижче по кваліфікаційному рівню на роботі, а також, що кількість працівників цієї ж спеціальності не менше 3.	Репозиторій основного сервісу	Worker	+
Від сервісу моніторингу приходять дані з датчика для запису в БД. Викинути виключення, якщо у репозиторіїї не існує сенсора з Id, вказаним в повідомленні.	Репозиторій основного сервісу	Measurement	+

Як видно з таблиці 4.25, багато операції потребують виконання у контексті вторинного потоку. Хоча бібліотека контекстуальної валідації наразі немає можливості роботи з асинхронними делегатами, проте даного функціоналу можна

досягти за допомогою «брудного хаку»: огорнути синхронний виклик метода Validate у делегат, та передати як параметр методу Task.Run.

4.3 Висновок до розділу

У даному розділі було описано програмно-апаратну систему, яка є середовищем використання бібліотеки контекстуальної валідації. Вона подібна до тих, що розробляються на реальних промислових проектах. Система складається із багатьог різношерстих модулів, сервісів, реалізованих за допомогою різних технологій та готових рішень. Ці компоненти системи виступають як контекст виконання операцій та відповідної перевірки інваріант, котрі можуть потребувати як синхронного, так і асинхронного виконання.

Загалом, даний розділ показав, що бібліотека однаково успішно працює із різними видами контексту:

- об’єкти класів в оперативній пам’яті в межах одного процесу;
- кеш в межах одного процесу;
- кеш як окремий процес;
- процес іншого веб-сервісу;
- реляційна СКБД.

5 СТАРТАП-ПРОЕКТ

Темою даної роботи є бібліотека контекстуальної валідації у середовищах .NET та .NET Core, проте у рамках даної роботи було також розроблено автоматизовану систему моніторингу за станом будівлі. Саме дана система і подається як стартап.

Станом на сьогоднішній день проблема витоку води у багатоквартирних будинках є досить актуальною. Протікання ініціюють руйнування конструкції будинку, завдання матеріальних збитків і шкоди здоров'ю для мешканців та просто створюють дискомфортні умови для життя. Особливо гостро дане питання стоїть для старих будівель, що знаходяться в аварійному стані саме через протічки, і нові можуть завдати їм катастрофічної шкоди.

Аби запобігти руйнування будинку слід своєчасно зафіксувати протікання. Наразі в Україні не існує спеціалізованих систем для вирішення даних проблем. Закордонні аналоги або мають високу вартість, або не у повній мірі підходять для даної задачі.

Результати даного проекту можна застосовувати у жилих багатоквартирних будинках, офісах та інших приміщеннях, що не мають у нормальному стані надмірного рівня вологості. Найкраще вищеописана система проявить себе при встановлені у нових багатоквартирних, ще не введених в експлуатацію, будинках із розхильними стяжками у приміщеннях, які підпадають під моніторинг.

5.1 Опис ідеї проекту

Таблиця 5.1 – Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Своєчасне виявлення протікань та інших відхилень стану будівлі від норми	ЖК, готелі	Зменшення витрат на ремонт через своєчасне виявлення відхілення

Продовження таблиці 5.1.

Зміст ідеї	Напрямки застосування	Вигоди для користувача
	Офіси	Зменшення витрат. Підвищення продуктивності працівників
	Приватні будинки	Зменшення витрат. Поліпшення стану здоров’я, якості життя.

Таблиця 5.2 – Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№ п/п	Техніко-економічні характеристики ідеї	(потенційні) товари/концепції конкурентів			W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Мій проект	ЖКГ	Датчики протікань			
1.	Своєчасність виявлення неполадки	1-2 хв	До місяця	10-15 хв	-	-	+
2.	Оперативність прибуття на місце	10-20 хв	Доба	1-2 хв	-	+	-
3.	Якість виконання робіт	10/10	5/10	5/10	-	+	-
4.	Аналіз накопичених даних про стан будівлі	Присутній	Відсутній	Відсутній	-	-	+

Продовження таблиці 5.2.

№ п/п	Техніко-економічні характеристики ідеї	(потенційні) товари/концепції конкурентів			W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Мій проект	ЖКГ	Датчики протікань			
5.	Вартість послуги	2 тис / міс	1 тис / міс	1 тис	+	-	-
6.	Мобільний додаток	Присутній	Відстуній	Відсутній	-	-	+
7.	Моніторинг в реальному часі	Присутній	Відстуній	Відсутній	-	-	+

5.2 Технологічний аудит ідеї проекту

Таблиця 5.3 – Технологічна здійсненність ідеї проекту

№ п/п	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Використовувати датчики для моніторингу стану будівлі	Мова програмування C, мікроконтроллери STM32	Наявні	Доступні
2	Використання серверу для обробки запитів від користувачів	Сервери Kestrel, IIS. Мова програмування C#, фреймворки ASP.NET, ASP.NET Core,	Наявні	Доступні

Продовження таблиці 5.3.

№ п/п	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
3	Прослуховування повідомлень від датчиків / контролерів	Сервер Kestrel / IIS. C# + WCF	Наявні	Доступні
4	Використання сучасного фронт-енду	Angular 2+, HTML, CSS, TypeScript	Наявні	Доступні
5	Своєчасна реакція на відхилення від норми. Автоматичне розподілення задач по працівниках. Сповіщення їх за допомогою мобільного додатку	Android / Java (Xamarin), C# / .NET	Наявні	Доступні
6	Аналіз накопичених даних для передбачень	MS SQL, C#, алгоритм експертного аналізу	Ненаявні	Недоступні
Обрана технологія реалізації ідеї проекту: STM32, ASP.NET, ASP.NET Core, WCF, Angular 2+, HTML, CSS, TypeScript, Android / Java (Xamarin)				

5.3.Аналіз ринкових можливостей запуску стартап-проекту

Таблиця 5.4 – Попередня характеристика потенційного ринку стартап-проекту

п/п	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців, од	10
2	Загальний обсяг продаж, грн/ум.од	100000
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу (вказати характер обмежень)	Зайнятість ринку гравцями. Бідність широкої аудиторії. Монополізація послуг існуючими гравцями
5	Середня норма рентабельності в галузі (або по ринку), %	20%

Таблиця 0.5 – Характеристика потенційних клієнтів стартап-проекту

№ п/п	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
1.	Своєчасне виявлення відхилень стану будівлі від норми	1. Приватний сектор 2. ЖК 3. ОСББ	Обсяг параметрів під моніторингом. Вимоги до якості	Підвищення адаптивності бізнесу Підвищення швикості надання інфраструктури

Продовження таблиці 5.5.

№ п/п	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
2.	Можливість самостійного моніторингу стану будівлі	1. Приватний сектор 2. ЖК 3. ОСББ	Обсяг параметрів під моніторингом.	Можливість гнучкого налаштування застосунків Конфігурація параметрів
3.	Зменшення витрат на експлуатації будівлі	1. Приватний сектор 2. ЖК 3. ОСББ	Об’єм коштів	Підвищення швикості реакції Зменшення часу починки

Таблиця 0.6 – Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1.	Крадіжка інтелектуальної власності	Крадіжка ідеї або ключової інтелектуальної інновації	Відсудження прав інтелектуальної власності Забезпечення якіснішого захисту інформації Зміна методики шифрування приватного ключа Попередження користувачів із подальшою співпрацею для мінімізації фактор загрози

Продовження таблиці 5.6.

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
2.	Отримання несанкціонованого доступу сторонніми особами	Хакерська атака, що може призвести до компроментації даних клієнтів	Залучення спеціалістів з інформаційної безпеки Використання засобів шифрування та резервного копіювання
3.	Відсутність ринку	Відсутність шляху збуту товару внаслідок помилкового орієнтування	Ретельний розгляд проблем потенційних клієнтів Залучення експертів та менторів Консультації із спеціалістами
4.	Недостача капіталовкладень	Витрачені усі кошти до моменту виходу на ринок	Пошук нових джерел інвестицій

Таблиця 0.7 – Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1.	Отримання інвестицій	Отримання капіталу що необхідний для реалізації продукту	Розробка продукту
2.	Успішна маркетингова політика	В результаті проведеної маркетингової політики отримана висока зацікавленість користувачів	Підтримка стабільної роботи системи та проведення масштабування системи Збільшення цін на використання сервісу Використання подібної маркетингової стратегії надалі для залучення нових користувачів

Продовження таблиці 5.7.

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
3.	Поглинання конкурентами	Пропозиція купівлі проекту або розроблених технологій одним із конкурентів	Розвиток розроблених технологій Оцінка вартості розроблених технологій

Таблиця 0.8 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
Олігополія	Незначна кількість конкурентів Велика ринкова сила Схожість використовуваних технологій	Інформування ринку щодо появи нового сервісу моніторингу
Галузевий	Загроза появи нових конкурентів Виркова влада споживачів Висока потреба у товарі	Інформування ринку щодо якості використовуваної новаторської технології Пропозиція гнучких цін
Внутрішньогалузева	Діяльність в одній галузі економіки Надання сервісів одного типу	Зменшення вартості сервісу Примноження каналів розподілу

Продовження таблиці 5.8.

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
Товарно-видова	Надання різних сервісів одного типу	Маркетингова політика
Цінова	Використання цін для покращення економічних умов збуту	Зменшення вартості платформи Використання нових каналів розподілу
Марочна	Пропозиція схожої платформи Спільна цільова аудиторія	Інформування ринку щодо появи нового сервісу моніторингу

Таблиця 0.9 – Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
1.	Унікальність сервісу	Розроблений продукт має унікальне співвідношення ціна / якість для свого цінового діапазону
2.	Цінова політика	Отримання прибутку здійснюється за рахунок гнучкої моделі оплати
3.	Модель “бізнес для бізнесу”	Бізнес модель грунтується на співпраці із готелями та ЖК. Даний підхід дозволить обійти цінову конкуренцію на ринку цільової аудиторії

Таблиця 0.10 – Порівняльний аналіз сильних та слабких сторін

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні						
			–3	–2	–1	0	+1	+2	+3
1.	Унікальність сервісу	14						+	
2.	Цінова політика	19							+
3.	Модель “бізнес для бізнесу”	13					+		

Таблиця 0.11 – SWOT- аналіз стартап-проекту

Сильні сторони: Якість та довготривалість Низькі ціни	Слабкі сторони: Нестача капіталовкладень Бізнес-модель залежить від політики окремих бізнесів
Можливості: Інвестиції Реалізація бізнес-моделі Розширений функціонал Висока зацікавленість цільової аудиторії	Загрози: Крадіжка інтелектуальної власності Компроментація даних клієнтів Відсутність ринку

5.4. Бізнес модель

Бізнес модель стартапу організована у вигляді Lean Canvas.

Коли ідея і концепція придумані, мета будь-якого проекту – сформувати вимоги до MVP (minimum viable product), зрозуміти, як буде виглядати продукт на початковому рівні, на чому заробляти і т.д.

Для запуску маркетплейса або будь-якого іншого стартапу знадобиться зібрати гіпотези бізнес-моделі в документі, який називається Lean Canvas. Мета Lean Canvas – визначити гіпотези для MVP.

5.4.1. Унікальна пропозиція

Рішення проблеми – автоматизований догляд за будинками. У початковий фокус попадає дрібна (проста у вирішенні), проте найбільш гостра проблема українського сегменту – протікання у будинках. Таким чином, стартап має зайняти свою нішу. При виникненні поломки, протікання та інших відхилень стану будівлі від норми, фахівець виїжджає на об'єкт і виконує необхідні дії по приведенню будівлі в норму. Таким чином, випливають наступні конкурентні переваги: власники будівель не переймаються за стан будинку, своєчасність виявлення поломок.

5.4.2. Клієнтський сегмент

Важливо прийняти обмеження у вигляді монополії ЖКГ на такий вид послуг. По ходу розширення ряду пропозицій клієнтура має розширюватися у наступному порядку:

- житлові комплекси, офісні центри, готелі та ОСББ, так як конкуренція тут менше і клієнти мають можливість зробити велике замовлення, зацікавлені в поліпшеннях;
- заможний приватний сектор, коли збільшиться товарообіг, собівартість буде нижче та з'явиться репутація у широких колах;
- корпоративні клієнти, коли з’явиться широкий спектр послуг.

5.4.3. Канали розповсюдження та поширення

Розкрутка продукту повинна відбуватися двома шляхами: віртуальним (через Інтернет) та фізичним (на конференціях та інших місцях концентрації зацікавлених у продукті сторін).

З огляду на прив'язку проектного рішення до конкретних будинків, розповсюдження продукції можливе тільки фізичним шляхом: фахівець повинен виїхати на об'єкт і оцінити необхідну роботу. Варто додати Інтернет калькулятор послуг.

5.4.4. Відносини з клієнтами

Накопичення клієнтської бази. Переконати власників ЖК, ОЦ, готелів та ОСББ у потребі в системі автоматизованого догляду за будинками. Особиста презентація по місцю, на конференціях тощо. Приватний сектор планується привертати шляхом «сарафанного радіо», реклами на місцях компактного проживання заможних громадян. Корпоративних клієнтів привертати на виставках, конференціях, шляхом широкої реклами.

Утримання клієнтської бази має відбуватися за рахунок якості (а не вартості) послуг, широкого вибору пакету послуг, продажу необхідного апаратно-програмного забезпечення частково (таким чином підсадити на «голку виробника»).

5.4.5. Канали виручки

Спочатку прямі продажі систем автоматизації та супровідного пакету послуг. Після закріплення на ринку – окремих комплектуючих. Далі експансія на суміжні (схожі за сферою) ринки (за прикладом CISCO), поки рано говорити які саме.

5.4.6. Ключові ресурси

– фінансові: інвестиції з хакатонів, бізнес-інкубаторів;

– фізичні: датчики, елементи керування, мікроконтролери, мережеве обладнання, сервер, інструменти для працівників ЖКГ;

– інтелектуальні: проект системи, клієнтська база;

– людські: програмісти, електрики, інженери.

Первинна команда може бути сформована у бізнес-інкубаторах, хакатонах. ЖКГ персонал найнятий пізніше.

5.4.7. Ключові партнери

– поставники мікроконтролерів, датчиків, елементів керування (склади, пізніше виробники закордонні, локальні за власними кресленнями);

– тренери ЖКГ-персоналу;

– агентства, котрі вивчають ринок ЖКГ-послуг;

– постійні клієнти (мережі готелів, крупні житлові комплекси, забудовники);

– виробники апаратного забезпечення для мікроконтролерів.

5.4.8. Ключові дії

– проектування системи, підбір технічної команди, створення прототипу, пошук інвесторів;

– найм ЖКГ-персоналу, їх тренінг та атестація, закупівля апаратного забезпечення, оренда серверу, пошук перших мали клієнтів;

– замовлення аналізу ринку, на основі якого має бути організована домовленість із першими постійними клієнтами;

– закупівля крупних партій обладнання від виробника;

– придбання власного виділеного серверу;

– проектування власного низькотехнологічного обладнання (датчики вологості тощо);

– закупівля у локального виробника по власному проекту, налагодження партнерства із виробниками;

– розширення штату працівників, перехід від горизонтальної до вертикальної ієрархії, створення нових підрозділів, вихід підприємства зі стадії стартапу.

5.4.9. Структура витрат

Найбільш дорогі ресурси: сервер, висококваліфіковані спеціалісти, крупні партії обладнання. Найбільш дорогі дії: проектування системи та обладнання.

5.5 Висновок до розділу

У даному розділі було сформовано ідею та зміст стартап-проекту системи моніторингу за станом будівель та виконано аналіз з точки зору макро- та мікроекономіки, а саме:

- визначено сильні, слабкі та нейтральні характеристики ідеї проекту;
- оцінена технологічна здійсненність ідеї проекту;
- оцінена попередня характеристика потенційного ринку стартап-проекту;
- викакано аналіз ринкових можливостей запуску стартап-проекту;
- оцінена характеристика потенційних клієнтів стартап-проекту;
- оцінені фактори загроз та можливостей;
- здійснено ступеневий аналіз конкуренції на ринку;
- обґрунтувано фактори конкурентоспроможності;
- зроблено порівняльний аналіз сильних та слабких сторін;
- виконано SWOT- аналіз стартап-проекту;
- побудовано гнучку бізнес-модель Lean Canvas.

ВИСНОВКИ

Результатами даної роботи є бібліотека контекстуальної валідації, яка являє собою Production Rule System та проект частини системи моніторингу за станом будівлі. Бібліотека є елементом науково-практичної новизни даної дисертації. Система моніторингу виступає як її середовище використання та предмет розділу Стартап-проект.

У роботі обґрунтовується вимога створення засобу контекстуальних верифікацій станів, який однаково успішно працював би з кожним із рекомендованих та популярних способів організації валідаційної логіки. За результатами проведеного в роботі аналізу існуючі фреймворки та бібліотеки неповнністю вирішують описану проблему, або взагалі її ігнорують.

Рішенням даної проблеми виступає описана у роботі бібліотека контекстуальної валідації. Вона організовує логіку перевірок за допомогою набору правил скомбінованих із валідаційних предикатів та делегатів реакції на їх порушення.

Отримана бібліотека може працювати у двох режимах:

- статична валідація;
- динамічна валідація.

Бібліотека має наступні переваги:

- призводить до зменшення об’єму коду валідації;
- може бути використана у підходах відкладеної валідації та завжди валідної сутності;
- має інтуїтивно зрозумілий API;
- враховує правила System.ComponentModel.DataAnnotations;
- реалізує Exception та Notification підходи;
- сприяє зменшенню цикломатичної складності;
- сприяє подоланню Code Smells;
- сприяє зменшенню Design Smells;
- гарантує безпеки типів;

– спрощує написання модульних тестів,

та недоліки:

– відсутня асинхронна валідація;

– можливий негативний вплив на продуктивність (не через синхронність);

– відсутня робота з декількома контекстами;

– не вирішена проблема контексту контексту;

– предикати не оформлені у шаблон Specification;

– відсутня робота із правилами FluentValidation;

– відсутня інтеграція із ASP.NET та ASP.NET Core.

У ході розробки було використано різноманітні техніки:

– патерни DSL;

– прийом EIMI;

– патерни GoF.

Бібліотека однаково успішно працює із різними видами контексту:

– об’єкти класів в оперативній пам’яті в межах одного процесу;

– кеш в межах одного процесу;

– кеш як окремий процес;

– процес іншого веб-сервісу;

– реляційна СКБД.

ПЕРЕЛІК ПОСИЛАНЬ

1. Validation in DDD [Електронний ресурс] / Leo Gorodinsky. – 2012. – Режим доступу до ресурсу: http://gorodinski.com/blog/2012/05/19/validation-in-domain-driven-design-ddd/

2. The Fallacy of Always Valid Entity [Електронний ресурс] / Jeffrey Palermo. – 2009. – Режим доступу до ресурсу: http://jeffreypalermo.com/blog/the-fallacy-of-the-always-valid-entity/

3. Always Valid Entity [Електронний ресурс] / Greg Young. – 2010. – Режим доступу до ресурсу: https://www.infoq.com/news/2009/01/greg-young-ddd

4. Fowler M. Anemic Domain Model [Електронний ресурс] / Martin Fowler. – 2003. – Режим доступу до ресурсу: https://www.martinfowler.com/bliki/AnemicDomainModel.html.

5. .NET Microservices: Architecture for Containerized .NET Applications [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/.

6. System.ComponentModel.DataAnnotations Namespace [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations?view=netframework-4.7.2.

7. Fowler M. Replacing Throwing Exceptions with Notification in Validations [Електронний ресурс] / Martin Fowler. – 2014. – Режим доступу до ресурсу: https://martinfowler.com/articles/replaceThrowWithNotification.html.

8. Fowler M. Specifications / M. Fowler, E. Evans., 2005. – 19 с.

9. Офіційний сайт пакетного менеджера NuGet. Statistics [Електронний ресурс] – Режим доступу до ресурсу: https://www.nuget.org/stats.

10. ValidationAttribute Class [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations.validationattribute?view=netframework-4.7.2.

11. Martin R. Design Principles and Design Patterns / Robert Martin., 2000. – 23 с.

12. Skinner J. Option to add a 'Context' to the Validate() method in the form of an object [Електронний ресурс] / Jeremy Skinner. – 2016. – Режим доступу до ресурсу: github.com/JeremySkinner/FluentValidation/issues/243.

13. Fowler M. Production Rule System [Електронний ресурс] / Martin Fowler. – 2009. – Режим доступу до ресурсу: https://martinfowler.com/dslCatalog/productionRule.html.

14. Fowler M. Domain-Specific Languages / M. Fowler, R. Parsons.. – 420 с. – (Addison-Wesley).

15. Design Patterns Elements of Reusable Object-Oriented Software / E.Gamma, R. Helm, R. Johnson, R. Vlissides., 2009. – 417 с. – (Addison-Wesley).

16. Pratt C. A Truly Generic Repository [Електронний ресурс] / Chris Pratt. – 2016. – Режим доступу до ресурсу: https://cpratt.co/truly-generic-repository/.

17. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языку C# – Санкт-Петербург: Питер, 2013. – 896 с. – (4).

18. Hunt A. The Pragmatic Programmer. From Journeyman to Master / A. Hunt, D. Thomas. – Berkeley: Addison-Wesley, 1999. – 352 с.

19. Троцький М. О. Бібліотека контекстуальної валідації у середовищі .NET / М. О. Троцький, Я. Ю. Дорогий // Інтеграція світових наукових процесів як основа суспільного прогресу / М. О. Троцький, Я. Ю. Дорогий. – Київ: Інститут інноваційної освіти, 2018. – С. 173–177.

20. Офіційний сайт пакетного менеджера NuGet. Statistics. Excel [Електронний ресурс] – Режим доступу до ресурсу: https://www.nuget.org/packages?q=excel

21. Maysh J. Benchmarking 5 popular .NET logging libraries [Електронний ресурс] / Jennifer Maysh. – 2016. – Режим доступу до ресурсу: https://www.loggly.com/blog/benchmarking-5-popular-net-logging-libraries/.

22. Provided Sinks [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/serilog/serilog/wiki/Provided-Sinks.